O'REILLY®

GraphQL

ЯЗЫК ЗАПРОСОВ ДЛЯ СОВРЕМЕННЫХ ВЕБ-ПРИЛОЖЕНИЙ

ПИТЕР®

Алекс Бэнкс, Ева Порселло

Learning GraphQL

Declarative Data Fetching for Modern Web Apps

Eve Porcello and Alex Banks

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Алекс Бэнкс, Ева Порселло

GraphQL

ЯЗЫК ЗАПРОСОВ ДЛЯ СОВРЕМЕННЫХ ВЕБ-ПРИЛОЖЕНИЙ

ПИТЕР®
Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

ББК 32.988.02-018
УДК 004.738.5
Б97

Бэнкс Алекс, Порселло Ева

Б97 GraphQL: язык запросов для современных веб-приложений. — СПб.: Питер, 2019. — 240 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1143-5

GraphQL — это язык запросов, альтернативный REST и ситуативным архитектурам веб-сервисов, самая революционная технология извлечения данных со времен Ajax. Точно как React изменил взгляд веб-разработчика на создание пользовательских интерфейсов, GraphQL полностью изменит практику передачи данных по HTTP. Это практическое руководство поможет вам приступить к работе с языком GraphQL.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492030713 англ.
Authorized Russian translation of the English edition of Learning GraphQL ISBN 9781492030713 © 2018 Eve Porcello and Alex Banks. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same
ISBN 978-5-4461-1143-5
© Перевод на русский язык ООО Издательство «Питер», 2019
© Издание на русском языке, оформление ООО Издательство «Питер», 2019
© Серия «Бестселлеры O'Reilly», 2019

Краткое содержание

Предисловие 10
Глава 1. Добро пожаловать в GraphQL 13
Глава 2. Теория графов 29
Глава 3. Анатомия запросов GraphQL 47
Глава 4. Схема GraphQL 78
Глава 5. API GraphQL 111
Глава 6. Клиенты GraphQL 161
Глава 7. GraphQL в реальном мире 196
Об авторах 238
Об иллюстрации на обложке 239

Оглавление

Предисловие 10

Благодарности 10

Условные обозначения, используемые в книге 11

Использование примеров кода 12

Глава 1. Добро пожаловать в GraphQL 13

Что такое GraphQL 14

Спецификация GraphQL 17

Принципы проектирования GraphQL 18

Происхождение GraphQL 18

История транспортировки данных 19

Удаленный вызов процедур 19

Простой протокол доступа к объектам 20

Архитектура REST 20

Недостатки архитектуры REST 21

Чрезмерная выборка данных 22

Недостаточная выборка данных 24

Управление конечными точками REST 26

GraphQL в реальном мире 27

Глава 2. Теория графов 29

Терминология теории графов 32

История теории графов 35

Деревья как графы ... 39
Графы в реальном мире ... 43

Глава 3. Анатомия запросов GraphQL ... 47

Инструменты API GraphQL ... 50
GraphiQL ... 50
GraphQL Playground ... 53
Открытые API GraphQL ... 54
GraphQL-запрос ... 55
Ребра и соединения ... 59
Фрагменты ... 61
Мутации ... 69
Подписки ... 71
Самодиагностика ... 73
Абстрактные синтаксические деревья ... 75

Глава 4. Схема GraphQL ... 78

Определение типов ... 79
Типы ... 79
Скалярные типы ... 81
Перечисления ... 81
Соединения и списки ... 82
Соединения «один к одному» ... 84
Соединения «один ко многим» ... 85
Соединения «многие ко многим» ... 87
Списки разных типов ... 89
Аргументы ... 93
Фильтрация данных ... 94
Мутации ... 98
Типы ввода ... 101
Возвращаемые типы ... 105
Подписки ... 106
Документация схемы ... 107

Глава 5. API GraphQL ... 111
Настройка проекта ... 112
Распознаватели ... 113
Корневые распознаватели ... 115
Распознаватели типов ... 117
Использование вводов и перечислений ... 122
Ребра и соединения ... 124
Пользовательские скаляры ... 130
Сервер apollo-server-express ... 135
Контекст ... 138
Установка MongoDB ... 139
Добавление базы данных к контексту ... 140
Авторизация с помощью аккаунта GitHub ... 143
Настройка GitHub OAuth ... 144
Процесс авторизации ... 146
Мутация githubAuth ... 147
Аутентификация пользователей ... 152
Резюме ... 159

Глава 6. Клиенты GraphQL ... 161
API GraphQL ... 161
Запросы на выборку ... 162
Инструмент graphql-request ... 163
Apollo Client ... 167
Apollo Client и React ... 168
Настройка проекта ... 169
Конфигурирование Apollo Client ... 169
Компонент Query ... 172
Компонент Mutation ... 177
Авторизация ... 179
Авторизация пользователя ... 180
Идентификация пользователя ... 185

Работа с кэшем ... 187
Политики выборки ... 187
Сохранение кэша ... 189
Обновление кэша ... 190

Глава 7. GraphQL в реальном мире ... 196
Подписки ... 197
Работа с подписками ... 197
Управление подписками ... 204
Выгрузка файлов ... 210
Обработка выгрузок на сервере ... 210
Публикация новых фотографий с помощью Apollo Client ... 212
Безопасность ... 221
Тайм-ауты запроса ... 221
Ограничения данных ... 222
Ограничение глубины запроса ... 223
Ограничение сложности запроса ... 226
Движок Apollo ... 229
Дальнейшее обучение ... 230
Инкрементная миграция ... 230
В первую очередь — разработка схемы ... 232
События GraphQL ... 234
Сообщество ... 235
Сообщество Slack Channels ... 236

Об авторах ... 238

Об иллюстрации на обложке ... 239

Предисловие

Благодарности

Эта книга не увидела бы свет без помощи многих феноменальных людей. Все началось с идеи Элли Макдональд (Ally MacDonald), редактора книги *Learning React*[1], которая побудила нас написать *Learning GraphQL*. Нам тогда очень повезло, что мы поработали с Алисией Янг (Alicia Young), которая подготовила книгу к печати. Выражаем благодарность Джастину Биллингу (Justin Billing), Мелани Ярброу (Melanie Yarbrough) и Крису Эдвардсу (Chris Edwards), которые отшлифовали все неточности в процессе чрезвычайно тщательного редактирования.

На протяжении работы над книгой нам посчастливилось получить отзывы Пегги Райзис (Peggy Rayzis) и Сашко Стубайло (Sashko Stubailo) из команды Apollo, которые поделились своими идеями и самой свежей информацией по поводу новейших возможностей. Спасибо также Адаму Ракису (Adam Rackis), Гарретту Маккалоу (Garrett McCullough) и Шиви Сингху (Shivi Singh), отличным техническим редакторам.

Мы написали эту книгу о GraphQL, потому что любим GraphQL. Надеемся, вы тоже.

---

[1] *Бэнкс А., Порселло Е.* React и Redux: функциональная веб-разработка. — СПб.: Питер, 2018.

Условные обозначения, используемые в книге

В этой книге используются следующие типографские условные обозначения.

Курсив

Курсивом выделяются новые термины, слова, на которых сделан акцент.

Рубленый шрифт

Обозначает URL-адреса и адреса электронной почты.

`Моноширинный шрифт`

Используется для текста (листингов) программ, а также внутри абзацев для выделения элементов программ: имен переменных или функций, названий баз данных, типов данных, имен переменных среды, инструкций и ключевых слов, имен файлов и каталогов.

`Моноширинный полужирный шрифт`

Обозначает команды или другой текст, который должен быть введен пользователем.

`Моноширинный курсивный шрифт`

Обозначает код, который должен быть заменен заданными пользователем значениями или значениями, определяемыми контекстом.

Этот элемент означает примечание.

Этот элемент означает предупреждение или предостережение.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) доступен для скачивания по адресу github.com/moonhighway/learning-graphql/.

Эта книга предназначена для оказания помощи в выполнении стоящих перед вами задач. Вы можете использовать код, приведенный в ней, в своих программах и документации. Вам не нужно обращаться к нам за разрешением, до тех пор пока вы не станете копировать значительную часть кода. Например, использование при написании программы нескольких фрагментов кода, взятых из данной книги, не требует специального разрешения. Но продажа и распространение компакт-диска с примерами из книг издательства O'Reilly — требует. Ответы на вопросы, в которых упоминаются материалы этой книги, и цитирование приведенных в ней примеров не требуют разрешения. Но включение существенного объема примеров кода, приводимых в данной книге, в документацию по вашему собственному продукту *требует* получения разрешения.

Ссылки на источник приветствуются, но не обязательны. В такие ссылки обычно включаются название книги, имя ее автора, название издательства и номер ISBN. Например: «GraphQL: язык запросов для современных веб-приложений», авторы Алекс Бэнкс, Ева Порселло. Copyright 2018 Eve Porcello, Alex Banks, 978-1-492-03071-3.

1 Добро пожаловать в GraphQL

Прежде чем королева Англии посвятила Тима Бернерса-Ли (Tim Berners-Lee) в рыцари, он был программистом. Работал в ЦЕРН, Европейской лаборатории физики элементарных частиц в Швейцарии, и был окружен рядами талантливых исследователей. Бернерс-Ли хотел помочь коллегам делиться их идеями, поэтому решил создать сеть, в которой ученые могли бы публиковать и обновлять информацию. В итоге проект стал первым веб-сервером и первым веб-клиентом, а браузер WorldWideWeb (позже переименованный в Nexus) был выпущен в ЦЕРН (www.w3.org/People/Berners-Lee/Longer.html) в декабре 1990 года.

Благодаря своему проекту Бернерс-Ли позволил исследователям просматривать и обновлять веб-контент на своих компьютерах. WorldWideWeb — это HTML, URL-адреса, браузер и интерфейс WYSIWYG («что видишь, то и получаешь») для обновления контента.

Сегодня Интернет не просто HTML в браузере. Интернет — это ноутбуки. Это умные часы. Это смартфоны. Это микросхема радиочастотной идентификации (RFID) в вашем гаджете. Это робот, который накладывает еду вашим кошкам, пока вы находитесь на работе.

Клиенты сегодня намного производительнее, но мы стремимся все к тому же: загружать данные как можно быстрее. Нам нужно, чтобы наши приложения были эффективными, потому что наши пользователи придерживаются высоких стандартов. Они ожидают, что наши приложения будут работать хорошо при любых условиях: от 2G на смартфонах до мощного оптоволоконного Интернета на компьютерах с большим экраном. Быстрые приложения упрощают

общение с нашим контентом. Быстрые приложения делают пользователей счастливыми. И да, быстрые приложения зарабатывают нам деньги.

Получение данных с сервера клиентом быстро и предсказуемо — это история Сети, ее прошлое, настоящее и будущее. Хотя в книге мы часто ссылаемся на минувшие годы, мы здесь для того, чтобы поговорить о современных решениях. Мы здесь, чтобы поговорить о будущем. Мы здесь, чтобы поговорить о GraphQL.

Что такое GraphQL

GraphQL (www.graphql.org) — это язык запросов для ваших API. А также среда выполнения для запросов с вашими данными. Сервис GraphQL является транспортно независимым, но обычно обслуживается через HTTP.

Чтобы продемонстрировать GraphQL-запрос и его ответ, взглянем на SWAPI (graphql.org/swapi-graphql/), API Star Wars. SWAPI является интерфейсом представления состояний (REST API), который был преобразован с помощью GraphQL. Мы можем использовать его для отправки запросов и получения данных.

GraphQL-запрос касается только необходимых данных. Рисунок 1.1, *слева*, представляет пример GraphQL-запроса. Мы запрашиваем данные для персоны принцессы Леи. И получаем запись, соответствующую Лее Органе, потому что указали, что нужны данные пятой персоны (`personID: 5`). Затем мы запрашиваем три поля данных: `name`, `birthYear` и `created`. *Справа* — полученный ответ: данные JSON отформатированы в соответствии с формой нашего запроса. Этот ответ содержит только те данные, которые нам нужны.

Затем мы можем настроить запрос, поскольку запросы интерактивны. Можно изменить его и увидеть новый результат. Если мы добавим поле `FilmConnection`, сможем запросить название каждого из фильмов с Леей, как показано на рис. 1.2.

Запрос вложен, и, когда он выполняется, можно перемещать связанные объекты. Благодаря этому требуется сделать один HTTP-запрос для двух типов данных. Нам не нужно совершать несколько

```
1  query {
2    person(personID:5) {
3      name
4      birthYear
5      created
6    }
7  }
```

```
{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "19BBY",
      "created": "2014-12-10T15:20:09.791000Z"
    }
  }
}
```

Рис. 1.1. Пользовательский запрос к API Star Wars

```
1  query {
2    person(personID:5) {
3      name
4      birthYear
5      created
6      filmConnection {
7        films {
8          title
9        }
10     }
11   }
12 }
```

```
{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "19BBY",
      "created": "2014-12-10T15:20:09.791000Z",
      "filmConnection": {
        "films": [
          {
            "title": "A New Hope"
          },
          {
            "title": "The Empire Strikes Back"
          },
          {
            "title": "Return of the Jedi"
          },
          {
            "title": "Revenge of the Sith"
          },
          {
            "title": "The Force Awakens"
          }
        ]
      }
    }
  }
}
```

Рис. 1.2. Запрос фильмов

подходов, чтобы развернуть несколько объектов. Мы не получаем дополнительных нежелательных данных об этих типах. С помощью GraphQL наши клиенты могут получить все необходимые данные по одному запросу.

Всякий раз, когда запрос выполняется на сервере GraphQL, он проверяется на наличие системы типов. Каждый сервис GraphQL определяет типы в схеме GraphQL. Вы можете представить систему

типов как схему данных вашего API, подкрепленную списком объектов, которые вы определяете. Например, запрос персоны поддерживается объектом `Person`:

```
type Person {
    id: ID!
    name: String
    birthYear: String
    eyeColor: String
    gender: String
    hairColor: String
    height: Int
    mass: Float
    skinColor: String
    homeworld: Planet
    species: Species
    filmConnection: PersonFilmsConnection
    starshipConnection: PersonStarshipConnection
    vehicleConnection: PersonVehiclesConnection
    created: String
    edited: String
}
```

Тип `Person` определяет все поля вместе с их типами, касающиеся принцессы Леи и доступные для запроса. В главе 3 мы подробно рассмотрим схему и систему типов GraphQL.

GraphQL часто упоминается как *декларативный* язык для извлечения данных. Под этим мы подразумеваем, что разработчики будут перечислять свои требования к данным, говоря о том, *какие* данные им нужны, и не уточняя, *как* они собираются их получить. Серверные библиотеки GraphQL существуют на разных языках, включая C#, Clojure, Elixir, Erlang, Go, Groovy, Java, JavaScript, .NET, PHP, Python, Scala и Ruby[1].

В этой книге мы сосредоточимся на том, как создавать сервисы GraphQL с помощью JavaScript. Все методы, которые мы обсуждаем в этой книге, можно применить к GraphQL, используя любой язык.

---

[1] См. GraphQL Server Libraries на странице graphql.org/code/.

Спецификация GraphQL

GraphQL — спецификация для клиент-серверного взаимодействия. Что делает спецификация? Она описывает возможности и характеристики языка. Мы пользуемся преимуществами языковых спецификаций, поскольку они обеспечивают наличие общего словаря и лучших практик использования языка сообществом.

Достойным внимания примером спецификации программного обеспечения является спецификация ECMAScript. Время от времени представители компаний — разработчики браузеров, технических компаний и сообщества в целом — собираются вместе и разрабатывают то, что должно быть включено в спецификацию ECMAScript (и исключено из нее). То же самое верно для GraphQL. Несколько людей собрались вместе и написали, что должно быть включено (и не включено) в язык. Теперь это служит руководством для всех реализаций GraphQL.

Когда спецификация была выпущена, создатели GraphQL также поделились ссылочной реализацией сервера GraphQL на JavaScript — `graphql.js` (github.com/graphql/graphql-js). Это полезный проект, но цель данной эталонной реализации заключается не в том, чтобы указать, какой язык вы используете для реализации своего сервиса. Это просто руководство. После того как вы поймете язык запросов и систему типов, вы сможете создать свой сервер на любом языке, который предпочитаете.

Если спецификация и реализация различны, то что на самом деле приведено в спецификации? Спецификация описывает язык и грамматику, которые вы должны применять при написании запросов. Она также устанавливает систему типов, механизмы ее выполнения и проверки. За исключением указанного, спецификация не особенно важна. GraphQL не определяет, какой язык использовать, как хранить данные или каких клиентов поддерживать. Язык запросов — это рекомендации, но фактический дизайн вашего проекта зависит от вас. (Если вы хотите подробнее разобраться во всем этом, можете изучить документацию по адресу facebook.github.io/graphql/.)

Принципы проектирования GraphQL

Несмотря на то что GraphQL не контролирует, как вы создаете свой API, спецификация предлагает некоторые рекомендации[1].

- *Иерархичность*. GraphQL-запрос является иерархическим. Поля встраиваются в другие поля, и запрос формируется подобно данным, которые он возвращает.
- *Ориентированность на продукт*. GraphQL управляется потребностями данных клиента, а также языком и временем выполнения, поддерживаемыми клиентом.
- *Строгая типизация*. Сервер GraphQL поддерживается системой GraphQL. В схеме каждая точка данных имеет определенный тип, насчет которого она будет проверена.
- *Запросы, определенные клиентом*. Сервер GraphQL предоставляет возможности, которые клиенты могут использовать.
- *Интроспектива*. Язык GraphQL может запрашивать серверную систему типов GraphQL.

Теперь, когда у вас есть базовое представление о спецификации GraphQL, поговорим о том, почему она была создана.

Происхождение GraphQL

В 2012 году сотрудники компании Facebook решили, что необходимо перестроить собственные мобильные приложения. Приложения iOS и Android компании были просто тонкими обертками представлений мобильного сайта. Компания Facebook использовала сервер RESTful и таблицы данных FQL (Facebook для SQL). Производительность была сомнительной, и приложения часто аварийно завершали работу. В этот момент инженеры поняли, что

---

[1] См. GraphQL Spec, июнь 2018 (http://facebook.github.io/graphql/June2018/#sec-Overview).

им необходимо улучшить способ передачи данных в клиентские приложения[1].

Ли Байрон (Lee Byron), Ник Шрок (Nick Schrock) и Дэн Шафер (Dan Schafer) намеревались пересмотреть данные со стороны клиента. Они решили создать GraphQL, язык запросов, который будет описывать возможности и требования моделей данных для клиентских/серверных приложений компании.

В июле 2015 года программисты выпустили первоначальную спецификацию GraphQL и эталонную реализацию GraphQL на JavaScript под названием `graphql.js`. В сентябре 2016 года спецификация GraphQL прошла стадию «технического предварительного просмотра». Это означало, что она была официально готова к выпуску, хотя уже много лет до этого применялась в Facebook. Сегодня GraphQL задействует почти все, что касается извлечения данных в компании Facebook, и используется в компаниях IBM, Intuit, Airbnb и др.

История транспортировки данных

GraphQL реализует некоторые новомодные идеи, но все это следует воспринимать в историческом контексте передачи данных. Задумываясь о передаче данных, мы пытаемся понять, как передавать их между компьютерами. Мы запрашиваем некоторые данные из удаленной системы и ожидаем ответа.

Удаленный вызов процедур

В 1960-х годах был изобретен удаленный вызов процедур (remote procedure call, RPC). Клиент инициировал RPC, который отправлял сообщение с запросом на удаленный компьютер, чтобы что-то сделать. Удаленный компьютер отправлял ответ клиенту. Эти компьютеры

---

[1] См. видео «Выборка данных для приложений React» Дэна Шафера (Dan Schafer) и Джинга Чена (Jing Chen), www.youtube.com/watch?v=9sc8Pyc51uU.

отличались от клиентов и серверов, которые мы используем сегодня, но процесс был в основном одинаковым: запрос некоторых данных у клиента, получение ответа от сервера.

Простой протокол доступа к объектам

В конце 1990-х годов в Microsoft был разработан протокол простого доступа к объектам (Simple Object Access Protocol, SOAP). SOAP использовал язык XML для кодирования сообщений и протокол HTTP для транспортировки. В протоколе SOAP также применялась система типов и была представлена концепция ресурсоориентированных вызовов для данных. SOAP давал довольно хорошие результаты, но вызвал разочарование, поскольку его реализации были довольно сложными.

Архитектура REST

Парадигма API, с которой вы, вероятно, наиболее знакомы сегодня, — REST. Архитектура REST была описана в 2000 году в докторской диссертации (bit.ly/2j4SIKI) Роя Филдинга (Roy Fielding) в Калифорнийском университете в Ирвайне. Филдинг описал ресурсоориентированную архитектуру, в которой пользователи будут перемещаться по веб-ресурсам, выполняя такие операции, как GET, PUT, POST и DELETE. Сеть ресурсов можно рассматривать как *виртуальную машину состояний*, а действия (GET, PUT, POST, DELETE) — как изменения состояния в машине. Сегодня мы можем считать такое само собой разумеющимся, но тогда это было довольно сложно. (Да, и Филдинг получил степень доктора наук.)

В архитектуре RESTful маршруты представляют информацию. Например, запрос информации от каждого из этих маршрутов даст конкретный ответ:

```
/api/food/hot-dog
/api/sport/skiing
/api/city/Lisbon
```

Архитектура REST позволяет создать модель данных с множеством конечных точек, это гораздо более простой подход, чем в предыдущих архитектурах. Архитектура предоставила новый способ обработки

данных во все более сложной сети, но не обеспечивала соблюдение конкретного формата ответов данных. Первоначально архитектура REST использовалась с XML. *AJAX* сперва было аббревиатурой, расшифровываемой как *асинхронный JavaScript и XML*, поскольку данные ответа из запроса Ajax были отформатированы на языке XML (теперь это автономное слово, записываемое как *Ajax*). Это был болезненный шаг для веб-разработчиков: приходилось анализировать ответы XML до того, как данные могли быть применены в JavaScript.

Вскоре после этого Дугласом Крокфордом (Douglas Crockford) была разработана и стандартизирована технология JavaScript Object Notation (JSON). JSON обеспечивает элегантный формат данных, который могут анализировать и использовать многие языки. Крокфорд написал книгу *JavaScript: The Good Parts*[1] (bit.ly/js-good-parts) (O'Reilly, 2008), в которой сообщил, что JSON — одна из отличных технологий.

Влияние архитектуры REST неоспоримо. Она используется для создания бесчисленных API. Разработчики различного звена только выиграли от ее применения. Существуют даже настолько проникшиеся форматом люди, яростно спорящие о том, что есть и чем не является RESTful, что они получили название *рестафарианцы*. Итак, если все так хорошо, почему Байрон, Шрок и Шафер опять собираются разрабатывать что-то новое? Ответ кроется в определенных недостатках архитектуры REST.

Недостатки архитектуры REST

Когда спецификация GraphQL увидела свет, некоторые позиционировали ее как замену REST. «REST мертва!» — воскликнули ее ранние последователи, а затем призвали всех нас бросить лопату в багажник и отвезти наши ничего не подозревающие API REST в лес. Это было отличным поводом для возбуждения интереса к тематическим блогам и начала диспутов на конференциях, но позиционирование GraphQL как убийцы REST — явное упрощение. Тонкий подход заключается в том, что по мере развития Всемирной паутины REST при определенных условиях проявляет признаки деформации. Язык GraphQL создавался для облегчения такой деформации.

---

[1] *Крокфорд Д.* JavaScript: сильные стороны. — СПб.: Питер, 2013.

Чрезмерная выборка данных

Предположим, что мы создаем приложение, которое использует данные из REST-версии SWAPI. В первую очередь нам нужно загрузить некоторые данные о персонаже номер 1, Люке Скайуокере[1]. Мы можем отправить GET-запрос, посетив сайт https://swapi.co/api/people/1/. Ответ — это данные в формате JSON:

```
{
  "name": "Luke Skywalker",
  "height": "172",
  "mass": "77",
  "hair_color": "blond",
  "skin_color": "fair",
  "eye_color": "blue",
  "birth_year": "19BBY",
  "gender": "male",
  "homeworld": "https://swapi.co/api/planets/1/",
  "films": [
    "https://swapi.co/api/films/2/",
    "https://swapi.co/api/films/6/",
    "https://swapi.co/api/films/3/",
    "https://swapi.co/api/films/1/",
    "https://swapi.co/api/films/7/"
  ],
  "species": [
    "https://swapi.co/api/species/1/"
  ],
  "vehicles": [
    "https://swapi.co/api/vehicles/14/",
    "https://swapi.co/api/vehicles/30/"
  ],
  "starships": [
    "https://swapi.co/api/starships/12/",
    "https://swapi.co/api/starships/22/"
  ],
  "created": "2014-12-09T13:50:51.644000Z",
```

[1] Обратите внимание, что данные SWAPI не включают самые последние фильмы о звездных войнах.

```
  "edited": "2014-12-20T21:17:56.891000Z",
  "url": "https://swapi.co/api/people/1/"
}
```

Это очень объемный ответ. Ответ, превышающий потребности наших приложений в данных. Нам просто нужна информация касательно имени, веса и роста:

```
{
  "name": "Luke Skywalker",
  "height": "172",
  "mass": "77"
}
```

Это явный случай *чрезмерной выборки* — мы получаем много данных, которые нам не нужны. Клиенту требуется три фрагмента данных, а мы возвращаем объект с 16 фрагментами, да еще и отправляем по сети информацию, которая бесполезна.

Как будет выглядеть этот запрос в GraphQL-приложении? Нам все еще нужно узнать имя, рост и вес Люка Скайуокера (рис. 1.3).

```
query {
  person(personID:1) {
    name
    height
    mass
  }
}
```

```
{
  "data": {
    "person": {
      "name": "Luke Skywalker",
      "height": 172,
      "mass": 77
    }
  }
}
```

Рис. 1.3. Запрос о Люке Скайуокере

Слева показан GraphQL-запрос. Мы запрашиваем только те поля, которые нам нужны. Справа мы получаем ответ в формате JSON, на этот раз содержащий только запрошенные данные без 13 лишних полей. Мы запрашиваем данные в определенной форме и получаем их обратно в той же форме. Ни больше ни меньше. Это более декларативно и, вероятно, позволит быстрее получить ответ, учитывая, что посторонние данные не извлекаются.

Недостаточная выборка данных

Наш менеджер проектов только что взглянул на монитор и хочет добавить еще одну функцию в приложение «Звездные войны». В дополнение к имени, росту и весу теперь нам нужно отобразить список названий фильмов, в которых есть такой герой, как Люк Скайуокер. После того как мы запросим данные по адресу https://swapi.co/api/people/1/, мы все равно должны выполнить дополнительные запросы для получения дополнительных данных. Это означает, что мы не сделали *достаточную* выборку данных.

Чтобы получить название каждого фильма, нам нужно иметь данные с каждого из маршрутов в массиве фильмов:

```
"films": [
  "https://swapi.co/api/films/2/",
  "https://swapi.co/api/films/6/",
  "https://swapi.co/api/films/3/",
  "https://swapi.co/api/films/1/",
  "https://swapi.co/api/films/7/"
]
```

Для получения этих данных требуется один запрос для Люка Скайуокера (https://swapi.co/api/people/1/), а затем еще пять для каждого из фильмов. Для фильма мы получаем еще один большой объект. На этот раз мы используем только одно значение.

```
{
  "title": "The Empire Strikes Back",
  "episode_id": 5,
  "opening_crawl": "...",
  "director": "Irvin Kershner",
  "producer": "Gary Kurtz, Rick McCallum",
  "release_date": "1980-05-17",
  "characters": [
    "https://swapi.co/api/people/1/",
    "https://swapi.co/api/people/2/",
    "https://swapi.co/api/people/3/",
    "https://swapi.co/api/people/4/",
    "https://swapi.co/api/people/5/",
    "https://swapi.co/api/people/10/",
```

```
    "https://swapi.co/api/people/13/",
    "https://swapi.co/api/people/14/",
    "https://swapi.co/api/people/18/",
    "https://swapi.co/api/people/20/",
    "https://swapi.co/api/people/21/",
    "https://swapi.co/api/people/22/",
    "https://swapi.co/api/people/23/",
    "https://swapi.co/api/people/24/",
    "https://swapi.co/api/people/25/",
    "https://swapi.co/api/people/26/"
  ],
  "planets": [
      //... Длинный список маршрутов
  ],
  "starships": [
      //... Длинный список маршрутов
  ],
  "vehicles": [
      //... Длинный список маршрутов
  ],
  "species": [
      //... Длинный список маршрутов
  ],
  "created": "2014-12-12T11:26:24.656000Z",
  "edited": "2017-04-19T10:57:29.544256Z",
  "url": "https://swapi.co/api/films/2/"
}
```

Если бы мы хотели перечислить персонажей из этого фильма, нам нужно было бы сделать гораздо больше запросов. В данном случае нам понадобятся еще 16 маршрутов и еще 16 обращений к клиенту. Каждый HTTP-запрос задействует клиентские ресурсы и приводит к передаче избыточных данных. В результате замедляется пользовательский интерфейс и пользователи с низкоскоростным подключением или медленными устройствами рискуют вообще не увидеть контент.

Решение проблемы недостаточной выборки с помощью GraphQL — определение вложенного запроса, а затем запрос всех данных в одной выборке, как показано на рис. 1.4.

```
1  query {
2    person(personID:1) {
3      name
4      height
5      mass
6      filmConnection {
7        films {
8          title
9        }
10     }
11   }
12 }
```

```
{
  "data": {
    "person": {
      "name": "Luke Skywalker",
      "height": 172,
      "mass": 77,
      "filmConnection": {
        "films": [
          {
            "title": "A New Hope"
          },
          {
            "title": "The Empire Strikes Back"
          },
          {
            "title": "Return of the Jedi"
          },
          {
            "title": "Revenge of the Sith"
          },
          {
            "title": "The Force Awakens"
          }
        ]
      }
    }
  }
}
```

Рис. 1.4. Подключение к фильмам

Здесь мы получаем только нужные данные в одном запросе. И, как всегда, форма запроса соответствует форме возвращаемых данных.

Управление конечными точками REST

Еще одна распространенная жалоба на API REST — отсутствие гибкости. Поскольку потребности в клиенте меняются, вам обычно приходится создавать новые конечные точки, и их количество может быстро выйти из-под контроля.

С API SWAPI REST нам приходилось запрашивать многочисленные маршруты. Большие приложения обычно используют настраиваемые конечные точки для минимизации HTTP-запросов. Вы можете увидеть, как появляются конечные точки, такие как `/api/character-with-movie-title`. Скорость разработки может снижаться, поскольку настройка новых конечных точек часто означает, что команды фронтенда и бэкенда имеют больше возможностей для планирования и общения друг с другом.

В случае GraphQL типичная архитектура включает в себя одну конечную точку. Единая конечная точка может выступать в качестве шлюза и управлять несколькими источниками данных, и одна конечная точка упрощает организацию данных.

В таком обсуждении недостатков REST важно отметить, что многие организации используют GraphQL и REST вместе. Настройка конечной точки GraphQL, которая извлекает данные из конечных точек REST, является вполне допустимым способом применения GraphQL. Это может быть отличным способом постепенного внедрения GraphQL в вашей организации.

GraphQL в реальном мире

Различные компании используют GraphQL для своих приложений, сайтов и API. Одним из наиболее заметных ранних пользователей GraphQL был сайт GitHub. Его REST API прошел три итерации, а версия 4 его публичного API применяет GraphQL. Как упоминалось на сайте developer.github.com/v4/, владельцы GitHub обнаружили, что «способность точно определять нужные данные — и только нужные вам данные — является большим преимуществом по сравнению с конечными точками REST API версии 3».

Другие компании, такие как The New York Times, IBM, Twitter и Yelp, также поверили в GraphQL, и разработчики из этих команд часто превозносят преимущества GraphQL на конференциях.

Проводятся по крайней мере три конференции, посвященные GraphQL: GraphQL Summit в Сан-Франциско, GraphQL Finland в Хельсинки и GraphQL Europe в Берлине. Сообщество поклонников технологии продолжает развиваться благодаря локальным встречам и конференциям.

Клиенты GraphQL. Как мы неоднократно говорили, GraphQL — это просто спецификация. Неважно, используете вы какие-либо методы с помощью React, или Vue, или JavaScript, или даже просто в браузере. В GraphQL есть рецепты по нескольким конкретным темам, но ваши архитектурные решения зависят от вас. Это привело к появлению инструментов для реализации некоторых приемов, выходящих за рамки спецификации. Встречайте клиентов GraphQL.

Клиенты GraphQL создаются для ускорения рабочего процесса групп разработчиков и повышения эффективности и производительности приложений. Они выполняют такие задачи, как сетевые запросы, кэширование данных и ввод данных в пользовательский интерфейс. Существует много клиентов GraphQL, но лидерами являются Relay (facebook.github.io/relay/) и Apollo (www.apollographql.com).

Relay — клиент компании Facebook, который поддерживает языки React и React Native. Relay — это посредник между компонентами React и данными, которые извлекаются с сервера GraphQL. Relay используется компаниями Facebook, GitHub, Twitch и т. д.

Клиент Apollo был разработан компанией Meteor Development Group под инициативой сообщества по созданию всеобъемлющего инструментария GraphQL. Клиент Apollo поддерживает все основные платформы фронтенд-разработки и независим от платформы. Apollo также реализует инструменты, которые помогают в создании сервисов GraphQL, повышении производительности бэкенд-сервисов и внедрении инструментов для мониторинга производительности API GraphQL. Компании, включая Airbnb, CNBC, The New York Times и Ticketmaster, используют Apollo Client в своих рабочих процессах.

Экосистема велика и продолжает меняться, но хорошей новостью является то, что спецификация GraphQL — довольно стабильный стандарт. В следующих главах мы обсудим, как написать схему и создать сервер GraphQL. Параллельно вы можете ознакомиться с учебными ресурсами для поддержки вашего путешествия, представленными в репозитории GitHub этой книги: github.com/moonhighway/learning-graphql/. Там вы найдете полезные ссылки, примеры и файлы.

Прежде чем мы перейдем к тактике работы с GraphQL, давайте немного поговорим о теории графов и богатой коллекции идей, приведших к созданию GraphQL.

2 Теория графов

Звучит сигнал об уведомлении. Вы добираетесь до телефона. Когда отключаете звук, видите два уведомления. Пятнадцать человек лайкнули твит, который вы написали вчера вечером. Прекрасно. Три человека ретвитнули его. Еще лучше. Ваша кратковременная популярность в Twitter представлена графом (как видно на рис. 2.1).

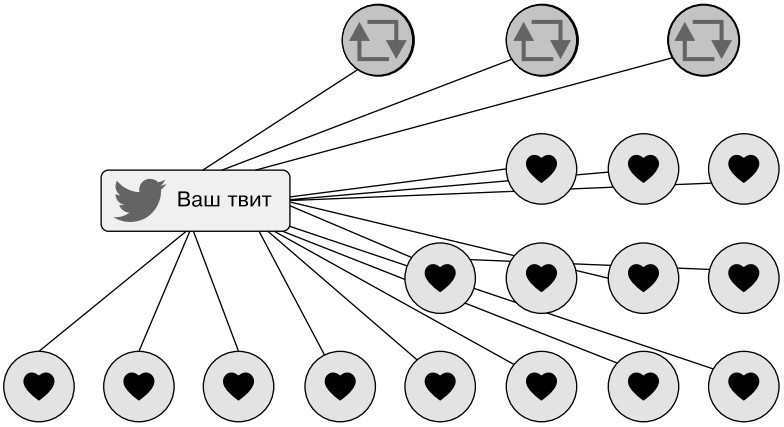

Рис. 2.1. Схема лайков и перепостов в Twitter

Вы поднимаетесь по лестнице, чтобы сесть в метро на станции «Ирвинг-парк». Вы запрыгиваете прямо перед закрытием дверей. Отлично. Поезд трясется на перегонах между станциями, пока вы едете в нем.

Двери открываются и закрываются на каждой станции. Сначала остановка «Эддисон». Затем «Паулина», «Саутпорт» и «Белмонт». В Белмонте вы переходите на другую платформу, чтобы пересесть на Красную линию. На Красной линии вы проезжаете еще две остановки: «Фуллертон» и «Север/Клибурн». Этот граф привел вас к тому, что показано на рис. 2.2.

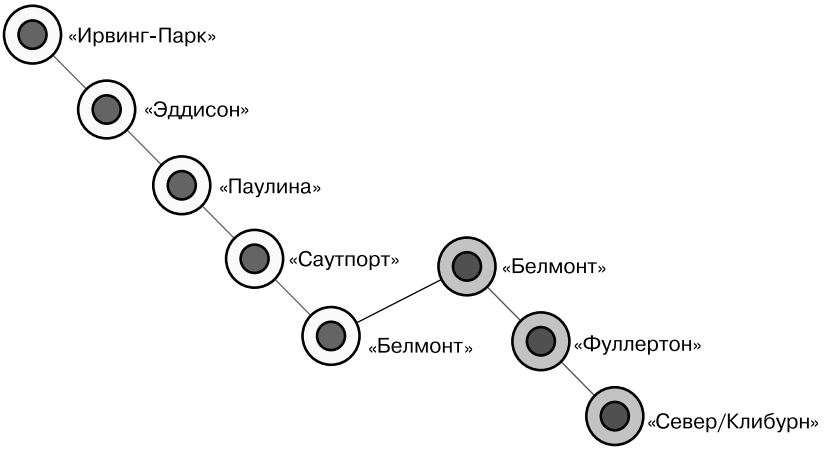

Рис. 2.2. Карта метрополитена Чикаго

Вы поднимаетесь по лестнице на улицу, и вдруг звонит ваш телефон. Это ваша сестра. Она говорит, что хочет купить билеты на поезд, чтобы в июле поехать на 80-летний юбилей своего дедушки. «По материнской или отцовской линии?» — спросите вы. «Отцовской, но я думаю, что родители мамы тоже там будут. И тетя Линда, и дядя Стив». Вы начинаете соображать, кто будет там. Как вариант, можно представить граф — генеалогическое древо. Рисунок 2.3 демонстрирует этот граф.

Вскоре вы начинаете замечать графы повсюду. Вы видите их в приложениях в социальных сетях, на картах маршрутов и заснеженных деревьях. От впечатляющих созвездий на небе, как видно на рис. 2.4. До самых маленьких структурных элементов природы, как на рис. 2.5.

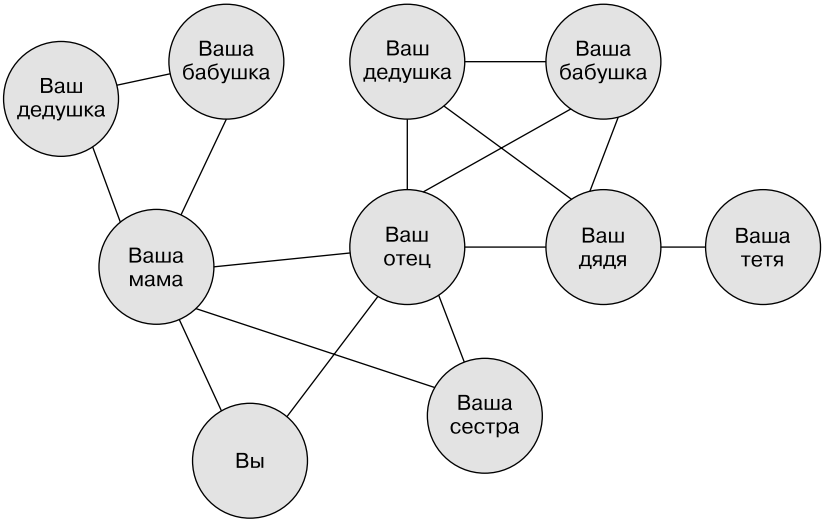

Рис. 2.3. Семейное древо

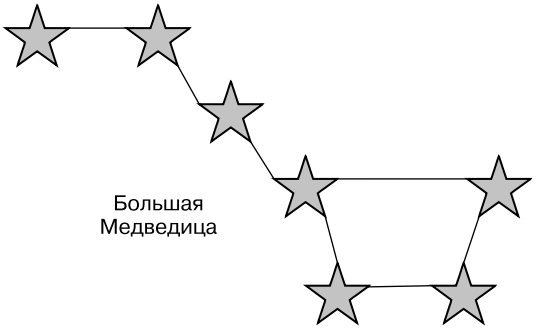

Рис. 2.4. Большая Медведица

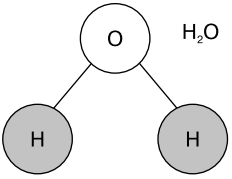

Рис. 2.5. Схематическое изображение молекулы воды (H_2O)

Графы повсюду вокруг нас, потому что представляют собой отличный способ для отображения взаимосвязанных элементов, людей, идей или фрагментов данных. Но откуда взялась концепция графов? Чтобы понять это, стоит более подробно рассмотреть теорию графов и ее истоки в математике.

Вам не нужно ничего знать о теории графов, чтобы успешно освоить GraphQL. Это не головоломка. Однако мы считаем, что интересно хотя бы узнать историю данного понятия.

Терминология теории графов

Теорию графов можно постигнуть непосредственно при их изучении. Графы используются формально для представления совокупности взаимосвязанных объектов. Вы можете представить граф как объект, содержащий точки данных и их соединения. В информатике графы обычно описывают сети данных. Граф может выглядеть примерно так, как показано на рис. 2.6.

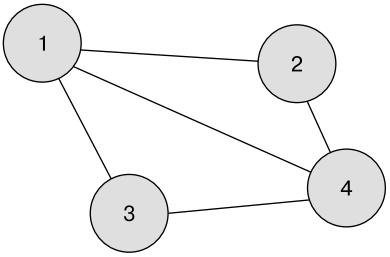

Рис. 2.6. Схематическое изображение графа

Эта схема графа состоит из четырех кругов, которые представляют точки данных. В терминологии графов они называются *вершинами* или *узлами*. Линии, или соединения, между этими вершинами называются *ребрами*, и всего их пять[1].

[1] Для дополнительного изучения узлов и ребер см. публикацию «Плавное введение в теорию графов» в блоге Вайдехи Джоши (Vaidehi Joshi), dev.to/vaidehijoshi/a-gentle-introduction-to-graph-theory.

Если выразить все как уравнение, граф — это $G = (V, E)$. Здесь G обозначает граф, а V — множество вершин, или узлов. Для данного графа V будет равным:

```
vertices = { 1, 2, 3, 4}
```

E обозначает множество ребер. Ребра представлены парами вершин:

```
edges = { {1, 2},
          {1, 3},
          {1, 4},
          {2, 4},
          {3, 4} }
```

Что произойдет, если в списке пар ребер мы изменим порядок? Например, так:

```
edges = { {4, 3},
          {4, 2},
          {4, 1},
          {3, 1},
          {2, 1} }
```

В этом случае граф останется таким же, как показано на рис. 2.7.

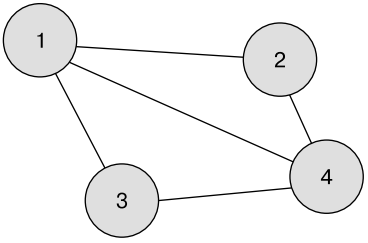

Рис. 2.7. Схематическое изображение графа

Уравнение все еще представляет граф, потому что между вершинами нет направления или иерархии. В теории графов это так называемый *неориентированный граф* (algs4.cs.princeton.edu/41graph/). Определения ребер, или соединения, между точками данных представляют собой *неупорядоченные пары*.

При перемещении по вершинам, или посещении различных вершин, этого графа вы можете начинать где угодно и заканчивать где угодно, двигаясь в любом направлении. Данные не следуют в явном перечисленном порядке, поэтому неориентированный граф является нелинейной структурой данных. Рассмотрим другой тип, *ориентированный граф*, который вы можете увидеть на рис. 2.8.

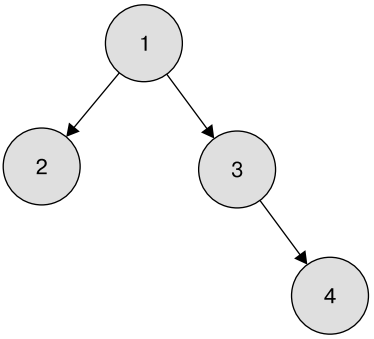

Рис. 2.8. Схематическое изображение ориентированного графа

Количество узлов одинаковое, но ребра выглядят иначе. Вместо линий они представлены стрелками. В этом графе есть направление, или поток, между вершинами. Чтобы представить его, мы использовали бы следующий код:

```
vertices = {1, 2, 3, 4}
edges = ( {1, 2},
          {1, 3}
          {3, 4} )
```

Сведя все вместе, мы можем представить уравнение графа таким образом:

```
graph = ( {1, 2, 3, 4},
          ({1, 2}, {1, 3}, {3, 4}) )
```

Обратите внимание, что пары заключены в круглые, а не в фигурные скобки. Скобки означают, что данные определения ребер являются упорядоченными парами. Всякий раз, когда ребра являются упорядоченными парами, мы имеем ориентированный граф, или

орграф. Что произойдет, если мы перестроим эти упорядоченные пары? Будет ли наша схема выглядеть так же, как в случае с неориентированным графом?

```
graph = ( {1, 2, 3, 4},
          ( {4, 3}, {3, 1}, {1, 2} ) )
```

Полученная схема теперь, с вершиной 4 в корне, будет выглядеть совсем по-другому, как показано на рис. 2.9.

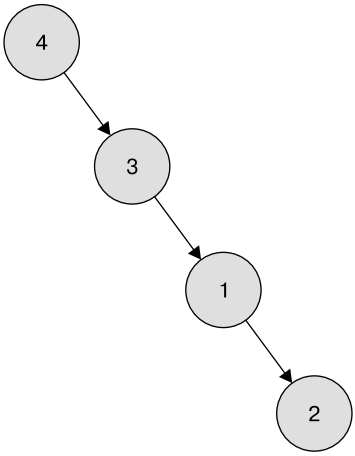

Рис. 2.9. Схематическое изображение ориентированного графа

Чтобы пройтись по графу, нужно будет начать движение с вершины 4 и посетить каждый узел графа, следуя по стрелкам. Для визуализации обхода можно представить физически перемещение с одной вершины на другую.

История теории графов

Мы начнем изучение истории теории графов с прусского города Кенигсберга (bit.ly/2AQhU47) в далеком 1735 году. Расположенный на реке Прегель, город был транспортным центром, который имел два больших острова, соединенных семью мостами с четырьмя другими островами, как показано на рис. 2.10.

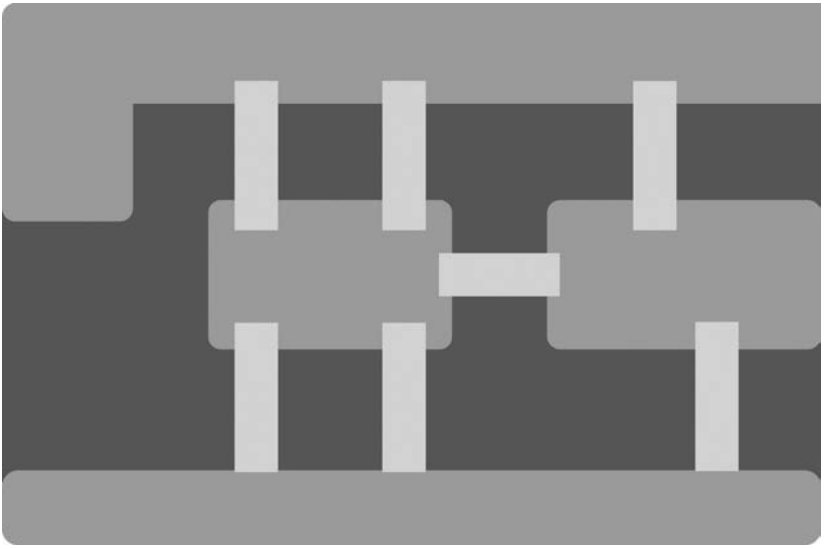

Рис. 2.10. Мосты Кенигсберга

Кенигсберг был великолепным городом, и его жители любили проводить воскресные дни на свежем воздухе и гулять по мостам. Со временем горожане всерьез задумались над решением головоломки: как пройти по каждому из семи мостов, не посещая ни один из них дважды? Они шли по городу, пытаясь посетить каждый остров и пересечь каждый мост, не повторяясь, но оказывались в тупике. Надеясь на помощь в решении этой проблемы, они обратились к Леонарду Эйлеру (Leonhard Euler). Эйлер был преуспевающим швейцарским математиком, который за всю свою жизнь опубликовал свыше 500 книг и статей.

Будучи гением, Эйлер не задумывался о такой ничтожной проблеме. Но после некоторых размышлений ученый так же, как и жители города, был заинтригован и попытался быстро отгадать эту загадку. Вместо того чтобы записывать все возможные пути, Эйлер решил, что было бы проще посмотреть на связи (мосты) между островами (рис. 2.11).

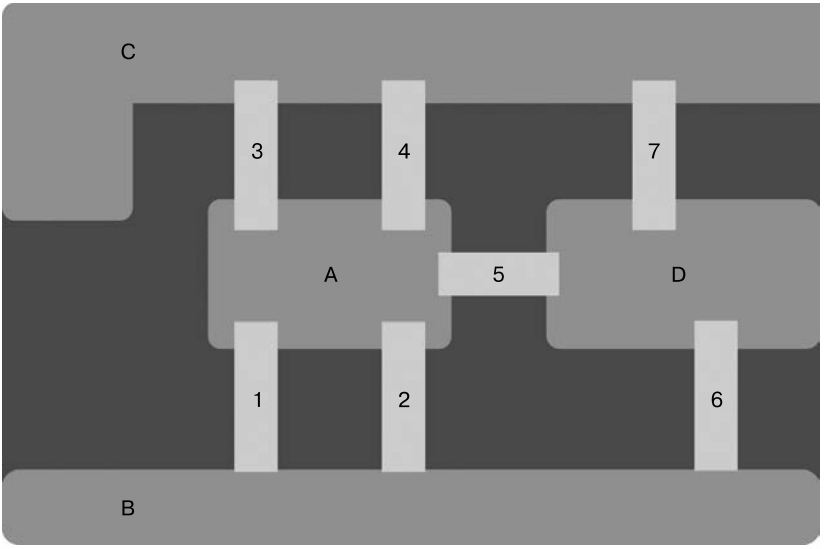

Рис. 2.11. Пронумерованные мосты Кенигсберга

Затем он упростил изображение, представив мосты и острова в виде схемы графа (рис. 2.12).

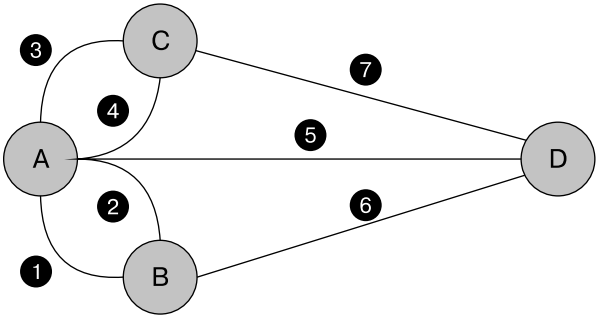

Рис. 2.12. Мосты Кенигсберга в виде схемы

На рис. 2.12 острова A и B *смежны*, потому что соединены ребрами. Используя эти граничные соединения, мы можем рассчитать

степень для каждой вершины. Степень вершины равна числу ребер, прикрепленных к ней. Если мы посмотрим на вершины в задаче о мостах, то обнаружим, что каждая из степеней нечетна.

- A — пять соединений с соседними вершинами (нечетное).
- B — три соединения с соседними вершинами (нечетное).
- C — три соединения с соседними вершинами (нечетное).
- D — три соединения с соседними вершинами (нечетное).

Поскольку каждая из вершин имела нечетные степени, Эйлер обнаружил, что пересечение каждого моста без повторного прохождения было невозможно. Иными словами, если вы посетите мост, чтобы добраться до острова, то должны уйти через другой мост. Количество ребер, или мостов, должно быть четным, если вы не хотите пересекать мост дважды.

Сегодня мы называем граф, в котором каждое ребро посещается единожды, *Эйлеровым*. Неориентированный граф будет иметь две вершины с нечетной степенью, или все вершины будут иметь четную степень. Здесь мы имеем две вершины с нечетной степенью (1, 4), как видно на рис. 2.13.

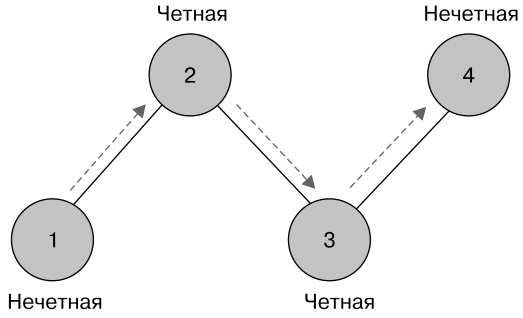

Рис. 2.13. Эйлеров путь

Другая идея, связанная с Эйлером, — это цепь, или *цикл Эйлера*. В данном случае начальная вершина совпадает с конечной. Каждое ребро посещается только один раз, но начальная и конечная вершины повторяются (рис. 2.14).

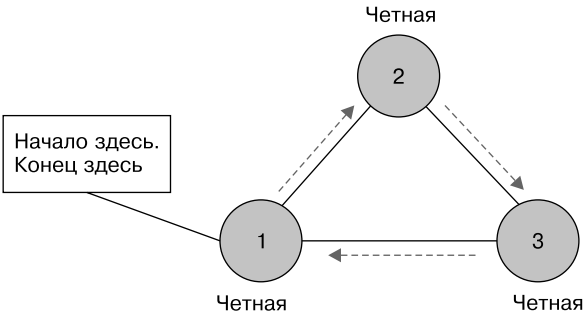

Рис. 2.14. Эйлеров цикл

Проблема кенигсбергских мостов стала первой теоремой теории графов. В дополнение к тому что Эйлер считается создателем теории графов, он известен созданием константы *e* и *мнимой единицы i*. Даже синтаксис математической функции *f(x)*, где функция *f* применяется к переменной *x*, можно проследить до Леонарда Эйлера[1].

Проблема кенигсбергских мостов состоит в том, что мост нельзя пересекать более одного раза. Никогда не было правила, согласно которому путешествие должно начинаться или заканчиваться на определенной вершине. Это означает, что попытка решить проблему была упражнением в неориентированном обходе графика. А как решить проблему с мостом, если нужно начать с определенной вершины?

Если вы живете на острове B, вам всегда нужно начинать путь с пересечениями. В этом случае вы будете иметь дело с ориентированным графом, обычно называемым деревом.

Деревья как графы

Рассмотрим другой тип графа — дерево. *Дерево* — это граф, в котором вершины расположены иерархически. Вы понимаете, что видите дерево, если есть корневая вершина. Другими словами, корень — начало дерева, а затем все остальные вершины связываются с ним как потомки.

---

[1] Более подробную информацию об Эйлере и его работе можно найти по адресу www.storyofmathematics.com/18th_euler.html.

Рассмотрим организационную схему. Это дерево издательства. Генеральный директор находится на вершине, а все остальные сотрудники — под ним. Генеральный директор — это *корень* дерева, а все остальные узлы — потомки корневой вершины, как показано на рис. 2.15.

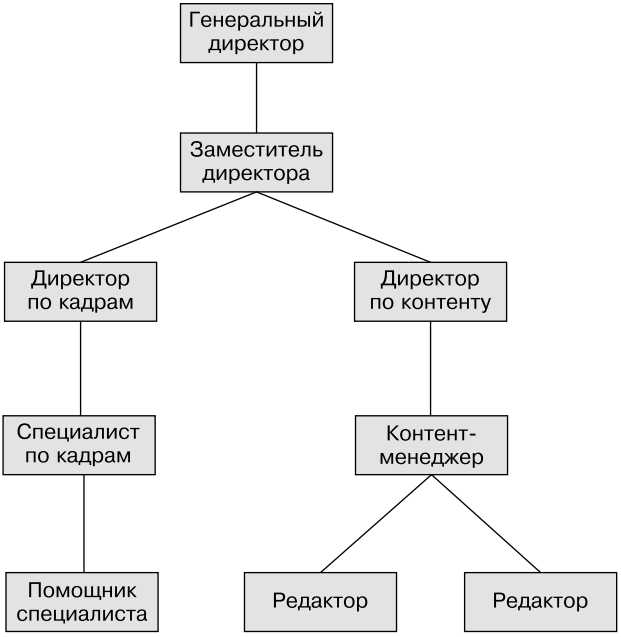

Рис. 2.15. Организационная схема

Деревья имеют много применений. Вы можете использовать дерево для создания генеалогии семьи. Деревья могут отражать алгоритмы принятия решений. Они помогают быстро и эффективно получать доступ к информации в базах данных. В один прекрасный день вам даже может понадобиться нарисовать на доске двоичное дерево из пяти человек, стоящих между вами и вашей новой работой.

Мы можем определить, является ли граф деревом, на основе того, имеет он корневую или начальную вершину. Из корневой вершины дерево связано ребрами с дочерними вершинами. Когда вершина

подключена к дочерней вершине, данный узел называется *родительским*. Когда у предка есть потомки, такая вершина называется *веткой*. Если вершина не имеет дочерних элементов, она называется *листом*.

Вершины содержат точки данных. Важно понять, где именно в дереве находятся данные, чтобы можно было оперативно получить доступ. Если необходимо быстро найти данные, следует рассчитать *глубину* отдельных вершин. Глубина вершины просто указывает, насколько удалена вершина от корня дерева. Рассмотрим дерево A –> B –> C –> D. Чтобы определить глубину вершины C, подсчитайте связи между C и корнем. Между C и корнем (A) существует ровно две связи, поэтому глубина вершины C равна 2, а глубина вершины D равна 3.

Иерархическая структура дерева означает, что деревья часто включают в себя другие деревья. Дерево, вложенное внутрь дерева, называется *поддеревом*. HTML-страница обычно имеет несколько поддеревьев. Корнем дерева является элемент `html`. Тогда есть два поддерева с элементом `head` в корне левого поддерева и элементом `body` в корне правого поддерева. Отсюда элементы `header`, `footer` и `div` — корни разных поддеревьев. С бóльшим количеством вложенности возникает много поддеревьев, как показано на рис. 2.16.

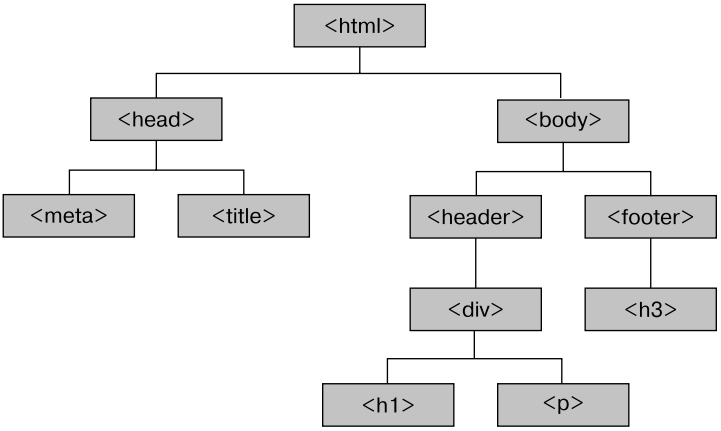

Рис. 2.16. Дерево HTML-страницы

Подобно тому как дерево является определенным типом графа, *двоичное (бинарное) дерево* является определенным типом дерева. Двоичное дерево означает, что каждая вершина имеет не более двух дочерних вершин. Говоря о двоичных деревьях, мы часто ссылаемся на *двоичные деревья поиска*[1]. Двоичное дерево поиска — это двоичное дерево, которое следует определенным правилам упорядочения. Правила упорядочения и древовидная структура помогают нам быстро найти нужные данные. На рис. 2.17 показан пример двоичного дерева поиска.

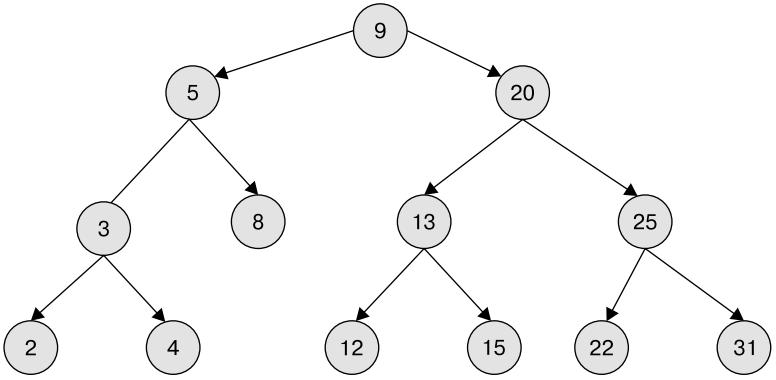

Рис. 2.17. Двоичное дерево поиска

Имеются корневая вершина и правило, что каждый узел должен иметь не более двух дочерних вершин. Предположим, что мы хотели найти вершину 15. Без двоичного дерева поиска нам нужно было бы перебирать каждую вершину до тех пор, пока мы не найдем вершину 15. Возможно, нам повезет и мы спустимся по правильной ветке. Но если нам не повезет, мы будем вынуждены неэффективно использовать дерево.

С деревом двоичного поиска мы можем быстро найти вершину 15, учитывая правило «право-лево». Если мы начнем обход

[1] См. публикацию «Листья двоичных деревьев» в блоге Вайдехи Джоши, bit.ly/2vQyKd5.

в корне (9), то скажем: «Пятнадцать больше или меньше девяти?» Если меньше, мы переместимся влево. Если больше — будем двигаться вправо. Пятнадцать больше девяти, так что мы будем двигаться вправо, и при этом мы исключим половину вершин дерева из поиска. Так у нас появляется вершина 20. Пятнадцать больше или меньше двадцати? Меньше, поэтому мы переместимся влево, исключая половину оставшихся вершин. Находясь на вершине 13, вычисляем, 15 больше или меньше 13? Больше, следовательно, мы пойдем вправо. Мы нашли! Используя правило «право-лево» в качестве фильтра, мы можем быстрее найти нужные данные.

Графы в реальном мире

Вы можете сталкиваться с концепциями теории графов каждый день в зависимости от работы, связанной с GraphQL. Или просто применять GraphQL как эффективный способ загрузки данных в пользовательские интерфейсы. Способ неважен, но все эти идеи прослеживаются в проектах GraphQL. Как мы видели, графы особенно хорошо подходят для удовлетворения потребностей разработки приложений с большим количеством точек данных.

Вспомните Facebook. Используя изученную терминологию, мы можем сказать, что каждый профиль в Facebook является вершиной. Когда профиль связан с другим профилем (друзья), на ребре показано двустороннее соединение. Facebook — неориентированный граф. Всякий раз, когда Ева связывается с кем-то в Facebook, они связываются с ней. Связь с ее лучшей подругой, Сарой, — двусторонняя. Они дружат друг с другом (рис. 2.18).

Рис. 2.18. Неориентированный граф Facebook

В качестве неориентированного графа каждая вершина в графе Facebook является частью сети взаимосвязанных отношений — социальной сети. Вы подключены ко всем своим друзьям. На том же графе эти друзья связаны со всеми своими друзьями. Обход может начинаться и заканчиваться на любой вершине (рис. 2.19).

Рис. 2.19. Неориентированная сеть Facebook

Есть также Twitter. В отличие от Facebook, где все вершины находятся в двустороннем соединении, Twitter является ориентированным графом, поскольку каждое ребро в нем одностороннее (рис. 2.20). Если вы будете следить за твитами Мишель Обамы, она может не следить за вами, даже если приглашать ее сделать это (@eveporcello, @moontahoe).

Рис. 2.20. Граф Twitter

Когда человек рассматривает все свои дружеские отношения, он становится корнем дерева. Он связан со своими друзьями. А его друзья связаны со своими друзьями в поддеревьях (рис. 2.21).

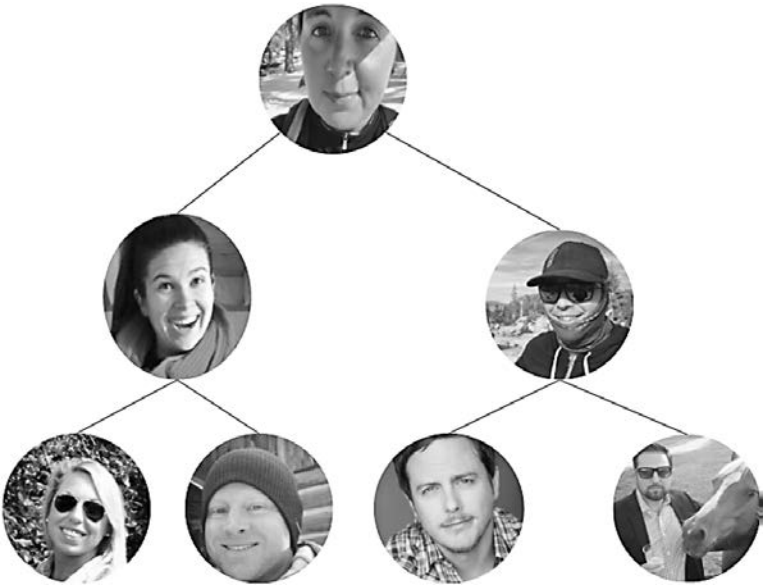

Рис. 2.21. Дерево друзей

То же самое было бы верно для всех пользователей в графе Facebook. Как только вы изолируете человека и запрашиваете его данные, запрос выглядит как дерево. Человек находится в корне, и все данные, которые требуется узнать об этой персоне, являются дочерней вершиной. В таком запросе человек привязан ко всем своим друзьям ребрами:

- персона:
  - имя;
  - местоположение;
  - день рождения;
  - друзья:
    - имя друга;
    - местоположение друга;
    - день рождения друга.

Эта структура очень похожа на GraphQL-запрос:

```
{
    me {
        name
        location
        birthday
        friends {
            name
            location
            birthday
        }
    }
}
```

С помощью GraphQL мы стремимся упростить сложные графы данных, выдавая запросы на необходимые нам данные. В следующей главе мы подробно рассмотрим, как работает GraphQL-запрос и как он проверяется на систему типов.

3 Анатомия запросов GraphQL

За 45 лет до того, как был создан GraphQL, сотрудник IBM Эдгар М. Кодд (Edgar M. Codd), выпустил краткую статью с очень длинным названием. «Реляционная модель данных для крупных общих банков данных» (bit.ly/2Ms7jxn), несмотря на название, содержала несколько мощных идей. В статье описана модель хранения и обработки данных с использованием таблиц. Вскоре после этого IBM начала работу над реляционной базой данных, которая может быть запрошена с применением *структурированного английского языка запросов — SEQUEL*, позже ставшего известным как SQL.

SQL, или Structured Query Language (структурированный язык запросов), — это язык, используемый для доступа к данным и управления ими в базе данных. SQL представил идею доступа к нескольким записям с помощью одной команды. Что также позволило получать доступ к любой записи с любым ключом, а не только с идентификатором.

Команды в SQL были интуитивно понятны: `SELECT`, `INSERT`, `UPDATE` и `DELETE`. Они описывают все, что вы можете сделать с данными. С помощью SQL мы можем написать один запрос, который позволяет возвращать связанные данные из нескольких таблиц данных в базе данных.

Идея в том, что указанные данные могут быть только прочитаны, созданы, обновлены или удалены, и это позволяет перейти к представлению Representational State Transfer (REST), что требует от нас использования разных HTTP-методов в зависимости от этих четырех основных операций с данными: GET, POST, PUT и DELETE. Однако

единственный способ указать желаемый тип данных для чтения или изменения с помощью REST — URL-адреса конечных точек, а не фактический язык запросов.

В GraphQL использованы идеи, которые изначально были разработаны для запросов к базам данных касательно Интернета. Один GraphQL-запрос может возвращать связанные данные. Подобно SQL, вы можете применять GraphQL-запросы для изменения или удаления данных. В конце концов, QL в SQL и в GraphQL означает одно и то же: язык запросов.

Несмотря на то что оба языка — это языки запросов, GraphQL и SQL совершенно разные. Они предназначены для совершенно разных сред. Вы отправляете SQL-запросы в базу данных. Вы отправляете GraphQL-запросы в API. Данные SQL хранятся в таблицах данных. Данные GraphQL можно хранить в любом месте в одной базе данных или нескольких файловых системах, API REST, веб-сокетах и других API GraphQL. SQL — это язык запросов для баз данных. GraphQL — язык запросов для Интернета.

GraphQL и SQL также имеют совершенно разный синтаксис. Вместо `SELECT` в GraphQL используется команда `Query` для запроса данных. Эта операция лежит в основе всего, что мы делаем с GraphQL. Вместо `INSERT`, `UPDATE` или `DELETE` GraphQL переносит все указанные изменения данных в один тип данных: `Mutation`. Поскольку GraphQL создан для Интернета, он включает тип `Subscription`, который можно применять для отслеживания изменений данных через соединения сокетов. SQL не имеет ничего подобного подписке. SQL как прародитель, который не похож на своего внука, но мы знаем, что они связаны друг с другом, потому что у них одна и та же фамилия.

GraphQL стандартизирован в соответствии со спецификацией. Неважно, какой язык вы применяете: GraphQL-запрос — это GraphQL-запрос. Синтаксис запроса — строка, которая выглядит одинаково независимо от того, используете вы проект на языке JavaScript, Java, Haskell или каком-то еще.

Запросы — это простые строки, которые отправляются в теле запросов POST в конечную точку GraphQL. Ниже приведен GraphQL-запрос, строка, написанная на языке запросов GraphQL:

```
{
  allLifts {
    name
  }
}
```

Вы отправили бы такой запрос в конечную точку GraphQL с помощью утилиты *curl*:

```
curl 'http://snowtooth.herokuapp.com/'
  -H 'Content-Type: application/json'
  --data '{"query":"{ allLifts {name }}"}'
```

Предполагая, что схема GraphQL поддерживает запрос в этой форме, вы получите ответ в формате JSON непосредственно в терминале. Указанный ответ JSON будет содержать либо данные, которые вы запрашивали в поле с именем `data`, либо поле `errors`, если что-то пошло не так. Мы делаем один запрос. Мы получаем один ответ.

Чтобы изменить данные, мы можем отправить *мутации*. Мутации очень похожи на запросы, но их задача состоит в том, чтобы что-то изменить в общем состоянии приложения. Данные, необходимые для выполнения изменения, могут быть отправлены непосредственно с мутацией, как показано здесь:

```
mutation {
  setLiftStatus(id: "panorama" status: OPEN) {
    name
    status
  }
}
```

Предыдущая мутация написана на языке запросов GraphQL, и мы можем предположить, что она изменит статус с идентификатором `panorama` на OPEN. Опять же мы можем отправить эту операцию на сервер GraphQL с помощью cURL:

```
curl 'http://snowtooth.herokuapp.com/'
  -H 'Content-Type: application/json'
  --data '{"query":"mutation {setLiftStatus(id: \"panorama\"
           status: OPEN) {name status}}"}'
```

Есть более привлекательные способы сопоставления переменных с запросом или мутацией, но мы расскажем об этих деталях позже в книге. В данной главе мы сосредоточимся на том, как создавать запросы, мутации и подписки с помощью GraphQL.

Инструменты API GraphQL

Сообщество GraphQL выпустило несколько инструментов с открытым исходным кодом, которые можно использовать для взаимодействия с API GraphQL. Данные инструменты позволяют писать запросы на языке GraphQL, отправлять эти запросы в конечные точки GraphQL и проверять ответ в формате JSON. Далее мы рассмотрим два наиболее популярных инструментария для тестирования запросов GraphQL через API GraphQL: среды GraphiQL и GraphQL Playground.

GraphiQL

GraphiQL — интегрированная браузерная среда разработки (integrated development environment, IDE), которая была создана в компании Facebook, чтобы вы могли запрашивать и исследовать API GraphQL. GraphiQL поддерживает подсветку синтаксиса, автозавершение кода и предупреждения об ошибках, а также позволяет запускать и просматривать результаты запроса непосредственно в браузере. Многие открытые API предоставляют интерфейс GraphiQL, с помощью которого вы можете запрашивать данные в реальном времени.

Интерфейс довольно прост. Используется панель, в которой вы пишете свой запрос, кнопка воспроизведения для его запуска и панель для отображения ответа, как показано на рис. 3.1.

Запросы пишутся на языке GraphQL. Мы рассматриваем этот текст как *документ запроса*. Вы размещаете текст запроса на левой панели. Документ GraphQL может содержать определения одной или нескольких *операций*. Операция — это `Query`, `Mutation` или `Subscription`. На рис. 3.2 показано, как добавить операцию `Query` в документ.

Рис. 3.1. Интерфейс GraphiQL

Рис. 3.2. GraphiQL-запрос

Запрос выполняется при нажатии кнопки воспроизведения. Затем на правой панели вы увидите ответ в формате JSON (рис. 3.3).

Вы можете щелкнуть в правом верхнем углу, чтобы открыть окно Docs, содержащее всю необходимую информацию, для взаимодействия с текущим сервисом. Эта документация автоматически добавляется в GraphiQL, поскольку она считывается из схемы сервиса. Схема определяет данные, доступные в сервисе, и GraphiQL автоматически создает документацию, запуская запрос интроспекции по схеме. Вы всегда можете изучить эту документацию на панели просмотра документации, как показано на рис. 3.4.

Рис. 3.3. GraphiQL

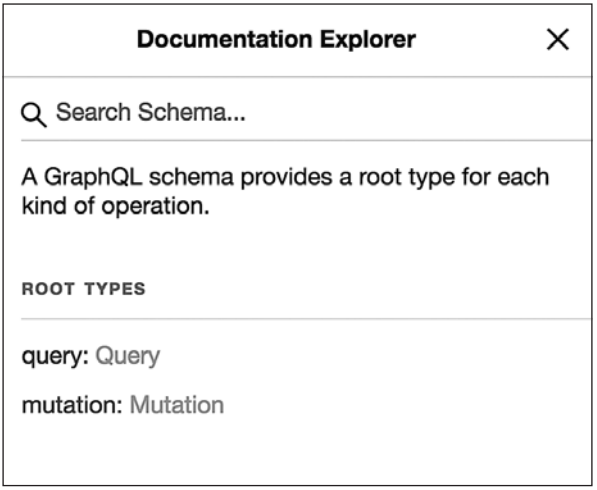

Рис. 3.4. Панель просмотра документации GraphiQL

Чаще всего вы будете обращаться к GraphiQL через URL-адрес, размещаемый рядом с самим сервисом GraphQL. Если вы создаете собственный сервис GraphQL, можете добавить маршрут, который

отобразит интерфейс GraphQL, чтобы ваши пользователи могли исследовать данные, публикуемые вами для них. Вы также можете загрузить отдельную версию GraphiQL.

GraphQL Playground

Еще один инструмент для изучения API GraphQL — GraphQL Playground. Созданная командой Prisma среда GraphQL Playground отражает функциональность GraphiQL и добавляет несколько интересных возможностей. Самый простой способ взаимодействия с GraphQL Playground — запустить ее в браузере на странице www.graphqlbin.com. После того как вы предоставите конечную точку, вы можете взаимодействовать с данными с помощью GraphQL Playground.

Среда GraphQL Playground очень похожа на GraphiQL, но в ней есть несколько дополнительных функций, которые могут оказаться удобными. Наиболее востребована из них возможность отправки пользовательских заголовков HTTP вместе с запросом GraphQL, как показано на рис. 3.5 (мы обсудим эту функцию более подробно, когда будем рассматривать авторизацию в главе 5).

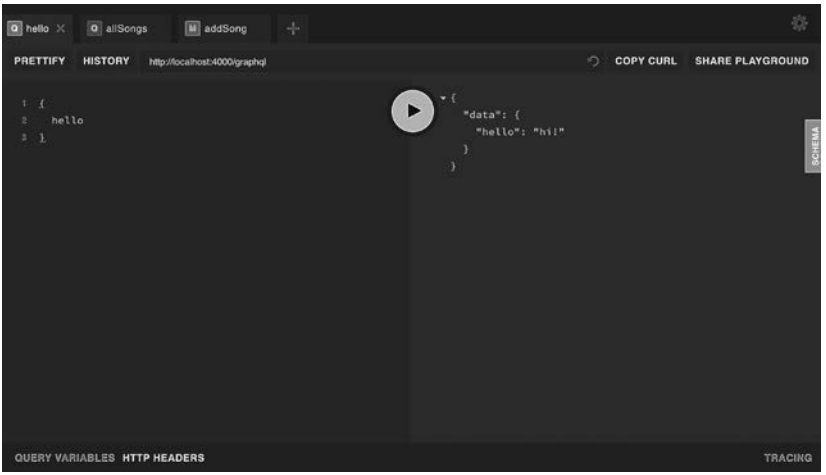

Рис. 3.5. GraphQL Playground

GraphQL Bin также является фантастическим инструментом для совместной работы, поскольку вы можете делиться ссылками на ваши бины с другими людьми, как показано на рис. 3.6.

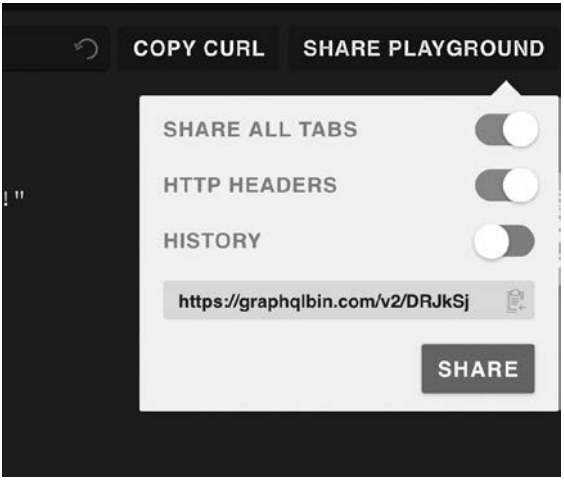

Рис. 3.6. Совместное использование бинов

Существует настольная версия GraphQL Playground, которую вы можете установить локально с помощью инструмента Home-brew:

```
brew cask install graphql-playground
```

Или можно загрузить дистрибутив с сайта github.com/prisma/graphql-playground/releases.

После того как вы установили среду или запустили инструмент GraphQL Bin, вы можете начать отправку запросов. Чтобы быстро приступить к работе, можно вставить конечную точку API в GraphQL Playground. Это может быть открытый API или ваш проект, запущенный на локальном хосте.

Открытые API GraphQL

Лучше всего начать работу с GraphQL с отправки запросов с использованием открытого API. Следующие компании предоставляют

интерфейс GraphiQL, который можно применять для запроса общедоступных данных.

- *SWAPI (the Star Wars API)* (graphql.org/swapi-graphql). Это проект Facebook, который является надстройкой API SWAPI REST.
- *GitHub API* (developer.github.com/v4/explorer/). Один из крупнейших открытых API GraphQL, GitHub GraphQL API позволяет отправлять запросы и мутации для просмотра и изменения ваших живых данных в GitHub. Чтобы взаимодействовать с данными, вам нужно войти в свою учетную запись GitHub.
- *Yelp* (www.yelp.com/developers/graphiql). Yelp поддерживает API GraphQL, который вы можете запросить с помощью GraphiQL. Вам нужно создать учетную запись разработчика Yelp для взаимодействия с данными в Yelp API.

Многие дополнительные примеры открытых API GraphQL доступны по адресу github.com/APIs-guru/graphql-apis.

GraphQL-запрос

«Снежный клык» (Snowtooth) — выдуманный горнолыжный курорт. Ради примеров в данной главе мы сделаем вид, что это настоящая гора и мы там работаем. Мы рассмотрим, как веб-команда «Снежного клыка» использует GraphQL для предоставления информации в режиме реального времени: сведений о состоянии подъемников и лыжных трасс. Лыжный патруль «Снежного клыка» может открывать и закрывать подъемники и трассы непосредственно со смартфона. Чтобы следовать примерам в этой главе, обратитесь к интерфейсу «Снежного клыка» на платформе GraphQL Playground (snowtooth.moonhighway.com/).

Вы можете использовать операцию `query` для запроса данных из API. Запрос описывает данные, которые вы хотите получить от сервера GraphQL. Когда вы отправляете запрос, вы запрашиваете единицы данных по *полям*. Эти поля отображаются в том же поле в ответе данных в формате JSON, который вы получаете с сервера. Например, если вы отправляете запрос `allLifts` и запрашиваете поля

`name` и `status`, вы должны получить ответ в формате JSON, содержащий массив `allLifts` и строки `name` и `status` каждого подъемника, как показано здесь:

```
query {
  allLifts {
    name
    status
  }
}
```

Обработка ошибок

Успешные запросы возвращают JSON-документ, содержащий ключ «данные». Неудачные запросы возвращают JSON-документ, содержащий ключ «ошибки». Детали того, что пошло не так, передаются в виде данных в формате JSON под этим ключом. Ответ JSON может содержать как «данные», так и «ошибки».

Вы можете добавить несколько запросов к документу запроса, но одновременно можно инициировать только одну операцию. Например, вы можете разместить две операции запроса в документе запроса:

```
query lifts {
  allLifts {
    name
    status
  }
}

query trails {
  allTrails {
    name
    difficulty
  }
}
```

Когда вы нажмете кнопку воспроизведения, среда GraphQL Playground предложит вам выбрать одну из этих двух операций.

Если требуется отправить один запрос всех указанных данных, вам нужно поместить все в один и тот же запрос:

```
query liftsAndTrails {
  liftCount(status: OPEN)
  allLifts {
    name
    status
  }
  allTrails {
    name
    difficulty
  }
}
```

Здесь налицо преимущества GraphQL. Мы можем получать разные типы данных в одном запросе. Мы запрашиваем `liftCount` по статусу, который позволяет нам узнать количество подъемников, имеющих в настоящее время этот статус. Мы также запрашиваем `name` и `status` каждого подъемника. Наконец, мы запрашиваем `name` и `status` каждой трассы.

`query` — это тип GraphQL. Мы называем его *корневым типом*, потому что это тип, который сопоставляется с операцией, а операции представляют собой корни нашего документа запроса. Поля, доступные для запроса в API GraphQL, определены в данной схеме API. В документации указывается, какие поля доступны для выбора в типе `query`.

Документация указывает нам, что мы можем выбирать поля `liftCount`, `allLifts` и `allTrails` при запросе этого API. Она также определяет больше полей, которые доступны для выбора, но весь смысл запроса заключается в том, что мы можем выбрать, какие поля нам нужны, а какие необходимо опустить.

Когда мы пишем запросы, мы выбираем поля, которые нам нужны, заключая их в фигурные скобки. Эти блоки называются *выборками*. Поля, которые мы определяем в выборке, напрямую связаны с типами GraphQL. Поля `liftCount`, `allLifts` и `allTrails` определены в типе `query`.

Вы можете встраивать множество выборок одну в другую. Поскольку поле `allLifts` возвращает список `Lift`, нам нужно использовать

фигурные скобки для создания новой выборки для этого типа. Есть все виды данных, которые мы можем запросить о подъемнике, но в указанном примере нам нужны только `name` и `status` подъемника. Аналогично запрос `allTrails` будет возвращать типы `Trail`.

Ответ в формате JSON содержит все данные, которые мы запросили. Эти данные форматируются как JSON и поставляются в том же виде, что и наш запрос. Каждому полю JSON выдается то же имя, что и поле в нашей выборке. Мы можем изменить имена полей в объекте ответа в запросе, указав псевдонимы, как показано ниже:

```
query liftsAndTrails {
  open: liftCount(status: OPEN)
  chairlifts: allLifts {
    liftName: name
    status
  }
  skiSlopes: allTrails {
    name
    difficulty
  }

}
```

Ниже приводится ответ:

```
{
  "data": {
    "open": 5,
    "chairlifts": [
      {
        "liftName": "Astra Express",
        "status": "open"
      }
    ],
    "skiSlopes": [
      {
        "name": "Ditch of Doom",
        "difficulty": "intermediate"
      }
    ]
  }
}
```

Теперь мы возвращаем данные в одну и ту же форму, но в нашем ответе мы переименовали несколько полей. Способ фильтрации результатов GraphQL-запроса состоит в передаче *аргументов запроса*. Аргументы — это пара значений ключа (или пары), связанная с полем запроса. Если требуются только имена закрытых подъемников, мы можем отправить аргумент, который будет фильтровать наш ответ:

```
query closedLifts {
    allLifts(status: "CLOSED" sortBy: "name") {
      name
      status
    }
}
```

Вы также можете использовать аргументы для выбора данных. Например, предположим, что нам нужно запросить статус отдельной канатной дороги. Мы можем выбрать подъемник по его уникальному идентификатору:

```
query jazzCatStatus {
    Lift(id: "jazz-cat") {
      name
      status
      night
      elevationGain
    }
}
```

Здесь мы видим, что ответ содержит `name`, `status`, `night` и `elevationGain` для канатной дороги Jazz Cat.

Ребра и соединения

В языке запросов GraphQL поля могут быть либо *скалярными типами*, либо *типами объектов*. Скалярные типы похожи на примитивы в других языках. Это листья наших выборок. В GraphQL имеется пять встроенных скалярных типов: целочисленные (`Int`), с плавающей запятой (`Float`), строки (`String`), логические (`Boolean`) и уникальные идентификаторы (`ID`). Как целые числа, так и числа

с плавающей запятой возвращают числа JSON, а строки и идентификаторы возвращают строки JSON. Логические типы просто возвращают логические значения. Несмотря на то что `ID` и `String` возвратят данные JSON того же типа, GraphQL все равно проверяет, возвращают ли идентификаторы уникальные строки.

Типы объектов GraphQL представляют собой группы из одного или нескольких полей, которые вы определяете в своей схеме. Они устанавливают форму объекта JSON, который должен быть возвращен. JSON может бесконечно встраивать объекты под полями, а также в GraphQL. Мы можем связать объекты вместе, запросив один объект для получения информации о связанных объектах.

Например, предположим, что нам нужно получить список маршрутов, к которым мы можем получить доступ с определенного подъемника:

```
query trailsAccessedByJazzCat {
    Lift(id:"jazz-cat") {
        capacity
        trailAccess {
            name
            difficulty
        }
    }
}
```

В предыдущем запросе мы запрашиваем некоторые данные о канатной дороге Jazz Cat. В нашу выборку входит запрос вместимости — поля `capacity`. Вместимость представляет собой скалярный тип; возвращается целое число, представляющее количество людей, которые могут одновременно ехать на одном подъемнике.

Поле `trailAccess` имеет тип `Trail` (тип объекта). В этом примере `trailAccess` возвращает отфильтрованный список маршрутов, доступных на Jazz Cat. Поскольку `trailAccess` является полем типа `Lift`, API может использовать информацию о родительском объекте Jazz Cat `Lift` для фильтрации списка возвращенных трасс.

В данном примере операция запрашивает соединение *«один ко многим»* между двумя типами данных, подъемниками и трассами. Один подъемник связан со многими связанными трассами. Если

мы начнем наш граф с узла `Lift`, то можем добраться до одного или нескольких вершин `Trail`, которые подключены к этому подъемнику через ребро с названием `trailAccess`. Чтобы наш граф считался неориентированным, нам нужно было бы вернуться к вершине `Lift` из узла `Trail`:

```
query liftToAccessTrail {
    Trail(id:"dance-fight") {
        groomed
        accessedByLifts {
            name
            capacity
        }
    }
}
```

В запросе `liftToAccessTrail` мы выбираем `Trail` под названием Dance Fight. Поле `groomed` возвращает логический скалярный тип, который позволяет нам узнать, подготовлена ли трасса Dance Fight. Поле `accessedByLifts` возвращает подъемники, которые доставляют лыжников на трассу Dance Fight.

Фрагменты

Документ GraphQL-запроса может содержать определения операций и *фрагментов*. Фрагменты — это выборки, которые можно повторно использовать в нескольких операциях. Взгляните на следующий запрос:

```
query {
    Lift(id: "jazz-cat") {
      name
      status
      capacity
      night
      elevationGain
      trailAccess {
        name
        difficulty
      }
```

```
    }
    Trail(id: "river-run") {
      name
      difficulty
      accessedByLifts {
        name
        status
        capacity
        night
        elevationGain
      }
    }
}
```

Этот запрос позволяет получить информацию о подъемнике Jazz Cat и трассе River Run. Выборка подъемника включает поля `name`, `status`, `capacity`, `night` и `elevationGain`. Информация, которую нам нужно получить о трассе River Run, включает подвыборку `Lift` для тех же полей. Мы могли бы создать фрагмент, который поможет уменьшить избыточность в нашем запросе:

```
fragment liftInfo on Lift {
  name
  status
  capacity
  night
  elevationGain
}
```

Фрагменты создаются с помощью идентификатора `fragment`. Фрагменты — это выборки определенных типов, поэтому вы должны указать тип, связанный с каждым фрагментом, в его определении. Фрагмент в данном примере называется `liftInfo`, и это выборка `Lift`.

Когда нужно добавить поля фрагмента `liftInfo` в другую выборку, следует указать три точки и имя фрагмента:

```
query {
    Lift(id: "jazz-cat") {
      ...liftInfo
      trailAccess {
        name
```

```
        difficulty
      }
    }
    Trail(id: "river-run") {
      name
      difficulty
      accessedByLifts {
        ...liftInfo
      }
    }
}
```

Синтаксис аналогичен синтаксису оператора `spread` языка JavaScript, который используется для аналогичной цели — назначения ключей и значений одного объекта другому. Эти три точки указывают GraphQL назначать поля из фрагмента текущей выборке. В данном примере мы можем выбрать `name`, `status`, `capacity`, `night` и `elevationGain` в двух разных позициях в нашем запросе, применяя один фрагмент.

Мы не смогли бы добавить фрагмент `liftInfo` в выборку `Trail`, поскольку она определяет только поля `Lift`. Мы можем добавить еще один фрагмент для трасс:

```
query {
    Lift(id: "jazz-cat") {
      ...liftInfo
      trailAccess {
        ...trailInfo
      }
    }
    Trail(id: "river-run") {
      ...trailInfo
      groomed
      trees
      night
    }
}

fragment trailInfo on Trail {
  name
  difficulty
```

```
  accessedByLifts {
    ...liftInfo
  }
}

fragment liftInfo on Lift {
  name
  status
  capacity
  night
  elevationGain
}
```

В этом примере мы создали фрагмент с именем `trailInfo` и использовали его в двух позициях в нашем запросе. Мы также применяем фрагмент `liftInfo` во фрагменте `trailInfo` для выборки сведений о подключенных подъемниках. Вы можете создать столько фрагментов, сколько нужно, и задействовать их взаимозаменяемо. В выборке, используемой в запросе River Run Trail, мы добавляем в наш фрагмент дополнительные детали, которые требуются по трассе River Run. Вы можете применять фрагменты в сочетании с другими полями в выборке. Вы также можете комбинировать несколько фрагментов одного типа в выборке:

```
query {
  allTrails {
    ...trailStatus
    ...trailDetails
  }
}

fragment trailStatus on Trail {
  name
  status
}

fragment trailDetails on Trail {
  groomed
  trees
  night
}
```

Что хорошо во фрагментах — вы можете модифицировать выборки, применяемые во многих разных запросах, путем изменения только одного фрагмента:

```
fragment liftInfo on Lift {
  name
  status
}
```

Это изменение в выборке во фрагменте `liftInfo` приводит к тому, что каждый запрос, который использует указанный фрагмент, выбирает меньше данных.

Объединения

Мы уже рассмотрели, как вернуть списки объектов, но в каждом случае до сих пор мы возвращали списки одного типа. Если вы хотите, чтобы список возвращал более одного типа, вы можете создавать *объединения*, связывающие разные типы объектов.

Предположим, мы разрабатываем планировщик для студентов, с помощью которого они смогут добавлять в повестку дня мероприятия, внеурочные занятия и семинары. Вы можете просмотреть пример на странице https://graphqlbin.com/v2/ANgjtr.

Если вы взглянете на документацию в GraphQL Playground, вы увидите, что объект `AgendaItem` является объединением, то есть он может возвращать несколько типов. В частности, `AgendaItem` вернет `Workout` или `StudyGroup`, которые могут быть частью расписания студента.

При написании запроса для повестки дня учащегося вы можете использовать фрагменты, чтобы определить, какие поля выбирать, когда `AgendaItem` является отработкой занятия (`Workout`), и какие поля выбрать, когда `AgendaItem` является семинаром (`StudyGroup`):

```
query schedule {
    agenda {
    ...on Workout {
      name
      reps
    }
```

```
    ...on StudyGroup {
      name
      subject
      students
    }
  }
}
```

Вот ответ:

```
{
  "data": {
    "agenda": [
      {
        "name": "Comp Sci",
        "subject": "Computer Science",
        "students": 12
      },
      {
        "name": "Cardio",
        "reps": 100
      },
      {
        "name": "Poets",
        "subject": "English 101",
        "students": 3
      },
      {
        "name": "Math Whiz",
        "subject": "Mathematics",
        "students": 12
      },
      {
        "name": "Upper Body",
        "reps": 10
      },
      {
        "name": "Lower Body",
        "reps": 20
      }
    ]
  }
}
```

Здесь мы используем *встроенные фрагменты*. У встроенных фрагментов нет имен. Они назначают выборки конкретным типам непосредственно в запросе. Мы применяем их для определения полей выборки, когда объединение возвращает разные типы объектов. Для каждой отработки занятия мы запрашиваем `name` и `reps` в возвращаемом объекте `Workout`. Для каждой группы мы запрашиваем `name`, `subject` и `students` в возвращаемом объекте `StudyGroup`. Возвращенная повестка будет состоять из одного массива, который содержит различные типы объектов.

Вы также можете задействовать именованные фрагменты для запроса объединения:

```
query today {
    agenda {
    ...workout
    ...study
  }
}

fragment workout on Workout {
  name
  reps
}

fragment study on StudyGroup {
  name
  subject
  students
}
```

Интерфейсы

Интерфейсы — это еще один вариант при работе с несколькими типами объектов, которые могут быть возвращены в одном поле. Интерфейс представляет собой абстрактный тип, устанавливающий список полей, которые должны быть реализованы в похожих типах объектов. Когда другой тип реализует интерфейс, он включает все поля из интерфейса и обычно некоторые из его собственных

полей. Если вы хотите попрактиковаться, посетите страницу https://graphqlbin.com/v2/yoyPfz.

Когда вы посмотрите на поле `agenda` в документации, вы увидите, что оно возвращает интерфейс `ScheduleItem`. Этот интерфейс определяет поля: `name`, `start time` и `end time`. Любой объект, который реализует интерфейс `ScheduleItem`, должен реализовать указанные поля.

В документации также сообщается, что типы `StudyGroup` и `Workout` реализуют данный интерфейс. Это означает, что мы можем с уверенностью предположить: оба указанных типа имеют поля `name`, `start` и `end`:

```
query schedule {
  agenda {
    name
    start
    end
  }
}
```

Запрос `schedule`, похоже, не заботится о том, чтобы поле `agenda` возвращало несколько типов. Ему нужны только имя, время начала и окончания для элемента, чтобы создать расписание того, когда и где должен быть этот студент.

При запросе интерфейса мы также можем использовать фрагменты для выборки дополнительных полей при возврате определенного типа объекта:

```
query schedule {
  agenda {
    name
    start
    end
    ...on Workout {
      reps
    }
  }
}
```

Запрос `schedule` был изменен, чтобы дополнительно запросить `reps`, когда `ScheduleItem` является `Workout`.

Мутации

До сих пор мы много говорили о чтении данных. Запросы касаются всех операций *чтения*, которые происходят в GraphQL. Чтобы записать новые данные, используются *мутации*. Мутации определяются как запросы. У них есть имена. Они могут иметь выборки, которые возвращают типы объектов или скаляры. Разница заключается в том, что мутации выполняют некое изменение данных, которое влияет на состояние исходных данных.

Например, опасная мутация будет выглядеть так:

```
mutation burnItDown {
  deleteAllData
}
```

`Mutation` — корневой тип данных. Схема API определяет поля, доступные для этого типа. API в предыдущем примере может уничтожить все данные клиента, выполнив поле с именем `deleteAllData`, которое возвращает скалярный тип: `true`, если все данные были успешно удалены и пришло время начать поиск нового задания, или `false`, если что-то пошло не так и требуется начать заново. Действительно ли данные будут удалены, зависит от реализации API, о которой мы поговорим далее, в главе 5.

Рассмотрим другую мутацию. Но вместо того, чтобы что-то уничтожить, давайте что-нибудь создадим:

```
mutation createSong {
  addSong(title:"No Scrubs", numberOne: true,
          performerName:"TLC") {
    id
    title
    numberOne
  }
}
```

Мы можем использовать этот пример для создания новых песен. Значения `title`, `numberOne` `status` и `per` `firstName` передаются в данную мутацию в качестве аргументов, и мы можем предположить, что мутация добавляет эту новую песню в базу данных. Если поле

мутации возвращает объект, вам нужно будет добавить выборку после мутации. В таком случае после его завершения мутация вернет тип `Song`, содержащий подробные сведения о только что созданной песне. Мы можем выбрать статус `id`, `title` и `numberOne` новой песни после мутации:

```
{
  "data": {
    "addSong": {
      "id": "5aca534f4bb1de07cb6d73ae",
      "title": "No Scrubs",
      "numberOne": true
    }
  }
}
```

Выше показан пример, как может выглядеть ответ на эту мутацию. Если что-то пошло не так, мутация вернет ошибку в ответе JSON вместо вновь созданного объекта `Song`.

Мы также можем задействовать мутации для изменения существующих данных. Предположим, что нам нужно изменить статус канатной дороги «Снежный клык». Мы могли бы использовать мутацию вот так:

```
mutation closeLift {
    setLiftStatus(id: "jazz-cat" status: CLOSED) {
     name
     status
  }
}
```

Мы можем применять эту мутацию, чтобы изменить статус подъемника Jazz Cat с открытого на закрытый. После мутации мы можем затем выбрать поля в `Lift`, которые были недавно изменены в нашей выборке. В таком случае мы получаем `name` подъемника — имя, которое было изменено, и новый статус — `status`.

Переменные запроса. До сих пор мы изменяли данные, посылая новые значения строк как аргументы мутации. В качестве альтернативы вы можете использовать *переменные*. Переменные заменяют статическое значение в запросе, чтобы вместо этого мы

могли передавать динамические значения. Рассмотрим нашу мутацию `addSong`. Вместо того чтобы иметь дело со строками, давайте применять имена переменных, которым в GraphQL всегда предшествует символ `$`:

```
mutation createSong($title:String! $numberOne:Int $by:String!) {
  addSong(title:$title, numberOne:$numberOne, performerName:$by)
  {
    id
    title
    numberOne
  }
}
```

Статическое значение заменяется переменной `$variable`. Затем мы утверждаем, что переменная `$variable` может быть принята мутацией. Оттуда мы сопоставляем каждое из имен переменных `$variable` с именем аргумента. В GraphiQL или Playground есть окно для переменных запроса. Здесь мы отправляем данные как объект JSON. Обязательно используйте правильные имена переменных в качестве ключей JSON:

```
{
  "title": "No Scrubs",
  "numberOne": true,
  "by": "TLC"
}
```

Переменные очень полезны при отправке данных в аргументе. Это не только упростит мутации в ходе тестирования, но и позволит динамически вносить изменения позже, при подключении клиентского интерфейса.

Подписки

Третий тип операции, доступный с помощью GraphQL, — это *подписка*. Бывают случаи, когда клиент может захотеть получать обновления в реальном времени с сервера. Подписка позволяет перехватывать события API GraphQL для изменения данных в реальном времени.

Подписка в GraphQL появилась в реальном времени в Facebook. Разработчики хотели отображать в режиме реального времени информацию о количестве лайков (Live Likes), которые получал пост, не обновляя страницу. Live Likes — это сценарий использования в режиме реального времени, основанный на подписках. Каждый клиент подписывается на подобное событие и видит, что его лайкнули, в реальном времени.

Подобно мутации и запросу, подписка является корневым типом. Изменения данных, которые клиенты могут отслеживать, определяются в схеме API как поля под типом `subscription`. Написание GraphQL-запроса для отслеживания подписки также похоже на то, как мы определяем другие операции.

Например, в приложении «Снежный клык» (snowtooth.moonhighway.com) мы можем отслеживать изменение статуса любого подъемника с помощью подписки:

```
subscription {
  liftStatusChange {
    name
    capacity
    status
  }
}
```

Когда мы запускаем эту подписку, мы отслеживаем изменения статуса подъемника через WebSocket. Обратите внимание, что при нажатии кнопки воспроизведения GraphQL Playgorund не сразу возвращает данные. Когда подписка отправляется на сервер, она отслеживает любые изменения данных.

Чтобы увидеть, что данные проходят к подписке, нам необходимо внести изменения. Нам нужно открыть новое окно или вкладку, чтобы отправить это изменение с помощью мутации. После того как операция подписки выполняется в окне среды GraphQL Playground, мы больше не можем запускать операции с применением того же окна или вкладки. Если вы используете GraphiQL для создания подписки, просто откройте второе окно браузера для интерфейса GraphiQL. Если вы используете GraphQL Playground, вы можете открыть новую вкладку, чтобы добавить мутацию.

Из нового окна или вкладки отправьте мутацию изменения статуса подъемника:

```
mutation closeLift {
    setLiftStatus(id: "astra-express" status: HOLD) {
      name
      status
  }
}
```

Когда мы запустим эту мутацию, статус подъемника Astra Express изменится и данные `name`, `capacity` и `status` подъемника Astra Express будут перенесены в нашу подписку. Astra Express — последний подъемник, который изменился, и новый статус присоединяется к подписке.

Изменим статус второго подъемника. Попробуйте установить статус подъемника Whirlybird как закрытый. Обратите внимание, что эта новая информация была передана нашей подписке. GraphQL Playground позволяет вам видеть оба набора ответных данных вместе с временем, когда данные были перенесены на подписку.

В отличие от запросов и мутаций, подписки остаются открытыми. Новые данные будут присоединяться к данной подписке каждый раз, когда на канатной дороге происходит смена статуса. Для прекращения отслеживания изменений статуса вам необходимо отказаться от подписки. Чтобы сделать это с помощью GraphQL Playground, просто нажмите кнопку остановки. К сожалению, единственный способ отказаться от подписки с помощью GraphiQL — закрыть вкладку браузера, на которой выполняется подписка.

Самодиагностика

Одной из самых мощных функций GraphQL является самодиагностика. *Самодиагностика* — это возможность запрашивать детали о текущей схеме API. Самодиагностика — это то, как элегантные документы GraphQL добавляются в интерфейс платформы GraphiQL.

Вы можете отправлять запросы каждому API GraphQL, который возвращает данные об указанной схеме API. Например, если нужно

знать, какие типы GraphQL доступны в «Снежном клыке», мы можем просмотреть эту информацию, выполнив запрос `__schema`, как показано ниже:

```
query {
  __schema {
    types {
      name
      description
    }
  }
}
```

Когда мы инициируем этот запрос, мы видим все типы, доступные в API, включая корневые, пользовательские и даже скалярные. Если нужны данные определенного типа, мы можем, выполняя запрос `__type`, отправить имя нужного типа в качестве аргумента:

```
query liftDetails {
  __type(name:"Lift") {
    name
    fields {
      name
      description
      type {
        name
      }
    }
  }
}
```

Этот запрос на самотестирование показывает нам все поля, доступные для запроса `Lift`. Когда вы познакомитесь с новым API GraphQL, неплохо было бы узнать, какие поля доступны для корневых типов:

```
query roots {
  __schema {
    queryType {
      ...typeFields
    }
    mutationType {
```

```
      ...typeFields
    }
    subscriptionType {
      ...typeFields
    }
  }
}

fragment typeFields on __Type {
  name
  fields {
    name
  }
}
```

Запрос самопроверки следует правилам языка запросов GraphQL. Избыточность предыдущего запроса была уменьшена за счет использования фрагмента. Мы запрашиваем имя типа и доступные поля каждого корневого типа. Самопроверка позволяет клиенту узнать, как работает текущая схема API.

Абстрактные синтаксические деревья

Документ запроса представляет собой строку. Когда мы отправляем запрос в API GraphQL, эта строка разбирается на *абстрактное синтаксическое дерево* и проверяется до запуска операции. Абстрактное синтаксическое дерево, или АСД, — иерархический объект, представляющий наш запрос. АСД — объект, который содержит вложенные поля, представляющие детали GraphQL-запроса.

Первым шагом в данном процессе является разделение строки на меньшие части. Это включает в себя анализ ключевых слов, аргументов и даже скобок и двоеточий в набор отдельных токенов. Такой процесс называется *лексированием* или *лексическим анализом*. Затем лексированный запрос анализируется в АСД. Запрос намного проще динамически модифицировать и проверять в виде АСД.

Например, ваши запросы начинаются как *документ* GraphQL. Документ содержит хотя бы одно *определение*, но оно также может содержать и список определений. Определения могут быть только

одного из двух типов: `OperationDefinition` или `FragmentDefinition`. Ниже приведен пример документа, который содержит три определения — две операции и один фрагмент:

```
query jazzCatStatus {
    Lift(id: "jazz-cat") {
      name
      night
      elevationGain
      trailAccess {
        name
        difficulty
      }
    }
}

mutation closeLift($lift: ID!) {
  setLiftStatus(id: $lift, status: CLOSED ) {
    ...liftStatus
  }
}

fragment liftStatus on Lift {
  name
  status
}
```

`OperationDefinition` может содержать только один из трех типов операций: `mutation`, `query` или `subscription`. Каждое определение операции содержит `OperationType` и `SelectionSet`.

Фигурные скобки, которые указываются после каждой операции, содержат выборку `SelectionSet`. Это фактические поля, которые мы запрашиваем вместе с их аргументами. Например, поле `Lift` — это `SelectionSet` для запроса `jazzCatStatus`, а поле `setLiftStatus` представляет собой выборку для мутации `closeLift`.

Выборки вложены друг в друга. Запрос `jazzCatStatus` содержит три вложенные выборки. Первая выборка `SelectionSet` включает поле `Lift`. Внутрь вложена выборка `SelectionSet`, которая содержит поля `name`, `night`, `elevationGain` и `trailAccess`. Ниже поля

`trailAccess` вложена еще одна выборка `SelectionSet`, которая включает поля `name` и `difficulty` для каждой трассы.

GraphQL может пройтись по этому АСД и проверить его относительно языка GraphQL и текущей схемы. Если синтаксис языка запроса правильный, а схема содержит поля и типы, которые мы запрашиваем, выполняется операция. Если нет, вместо этого возвращается соответствующая ошибка.

Кроме того, данный объект АСД легче модифицировать, чем строку. Если бы мы хотели добавить количество открытых подъемников в запрос `jazzCatStatus`, мы могли бы сделать это, напрямую изменив АСД. Все, что нам нужно сделать, — добавить дополнительную выборку `SelectionSet` к операции. АСД являются неотъемлемой частью GraphQL. Каждая операция анализируется в АСД, чтобы ее можно было проверить и в конечном итоге выполнить.

В данной главе вы узнали о языке запросов GraphQL. Теперь мы можем использовать его для взаимодействия с сервисом GraphQL. Но это было бы невозможно без конкретного определения того, какие операции и поля доступны для конкретного сервиса GraphQL. Это конкретное определение называется *схемой GraphQL*, и в следующей главе мы подробно рассмотрим, как создавать схемы.

4 Схема GraphQL

GraphQL изменит ваш процесс проектирования. Вместо представления API в виде группы конечных точек REST вы начнете рассматривать свои API как коллекции типов. Прежде чем переделывать API, вам нужно подумать, обговорить и формально определить типы данных, которые он будет использовать. Такая коллекция типов называется *схемой*.

Schema First — это методология проектирования, при которой все ваши команды оповещаются о типах данных, составляющих ваше приложение. Команда бэкенд-разработчиков будет иметь четкое представление о данных, которые необходимо хранить и доставлять. У команды фронтенд-разработки будут определения, необходимые для создания пользовательских интерфейсов. У каждого будет четкий список терминов, которые они смогут применять для общения относительно собираемой системы. Одним словом, каждый получит свою долю работы.

Чтобы упростить определение типов, GraphQL поддерживает язык, который можно использовать для определения схем, называемый *Schema Definition Language* или SDL. Так же как и язык запросов GraphQL, SDK GraphQL не зависит от того, какой язык или структуру вы задействуете при разработке своих приложений. Файлы схемы GraphQL представляют собой текстовые документы с определениями типов, доступных в приложении, и впоследствии используются клиентами и серверами для проверки запросов GraphQL.

В этой главе мы рассмотрим SDK GraphQL и скомпонуем схему приложения для обмена фотографиями.

Определение типов

Лучший способ разобраться с типами и схемами GraphQL — создать схему самому. Приложение для обмена фотографиями позволит пользователям авторизовываться через свои учетные записи GitHub, чтобы публиковать фотографии и отмечать пользователей на этих фотографиях. Управление пользователями и сообщениями представляет собой функциональность, лежащую в основе веб-приложений почти любого типа.

Приложение PhotoShare будет иметь два основных типа: `User` и `Photo`. Приступим к разработке схемы для всего приложения.

Типы

Основным элементом любой схемы GraphQL является тип. В GraphQL тип представляет собой пользовательский объект, и эти объекты описывают основные функции вашего приложения. Например, приложение для социальных сетей состоит из пользователей и сообщений (`Users` и `Posts`). Блог будет состоять из категорий и статей (`Categories` и `Articles`). Типы представляют данные вашего приложения.

Если вы создаете сервис типа Twitter, `Post` будет содержать текст, который пользователь хочет транслировать. (В этом случае для данного типа лучше всего подойдет имя `Tweet`.) Если вы работаете над сервисом Snapchat, `Post` будет содержать изображение и подойдет имя `Snap`. При определении схемы вы определяете общий язык, который ваша команда будет применять в разговоре о ваших доменных объектах.

Тип имеет *поля*, которые представляют данные, связанные с каждым объектом. Каждое поле возвращает определенный тип данных.

Это может быть как целое число или строка, так и пользовательский тип объекта или список типов.

Схема представляет собой набор определений типов. Вы можете указать свои схемы в файле JavaScript в виде строки или в любом текстовом файле. Обычно эти файлы имеют расширение `.graphql`.

Определим первый тип объекта GraphQL в нашем файле схемы — `Photo`:

```
type Photo {
    id: ID!
    name: String!
    url: String!
    description: String
}
```

Между фигурными скобками мы определили поля типа `Photo`. Поле `url` объекта `Photo` представляет собой ссылку на местоположение файла изображения. Это описание также содержит некоторые метаданные об объекте `Photo`: `name` и `description`. Наконец, каждый объект `Photo` будет иметь `ID`, уникальный идентификатор, который можно использовать в качестве ключа для доступа к фотографии.

Каждое поле содержит данные определенного типа. Мы установили только один пользовательский тип в нашей схеме, `Photo`, но GraphQL содержит несколько встроенных типов, которые мы можем задействовать для наших полей. Эти встроенные типы называются *скалярными*. В полях `description`, `name` и `url` применяется скалярный тип `String`. Данные, возвращаемые при запросе таких полей, будут строками в формате JSON. Восклицательный знак указывает, что данное поле не может быть *нулевым* (`null`), а это означает, что поля `name` и `url` должны возвращать некоторые данные в каждом запросе. Описание (`description`) может быть *нулевым*, следовательно, описания фотографий являются необязательными. При запросе это поле может вернуть значение `null`.

Поле `ID` объекта `Photo` определяет уникальный идентификатор для каждой фотографии. В GraphQL используется скалярный тип `ID`, когда нужно возвращать уникальный идентификатор. Значение JSON для этого идентификатора — строка, которая будет проверена на уникальность.

Скалярные типы

Встроенных скалярных типов GraphQL (`Int`, `Float`, `String`, `Boolean` и `ID`) будет в большинстве случаев достаточно, но иногда вам могут понадобиться собственные скалярные типы. Скалярный тип не является объектом. У него нет полей. Однако при реализации сервиса GraphQL вы можете указать, как пользовательские скалярные типы должны проверяться на допустимость, например:

```
scalar DateTime

type Photo {
    id: ID!
    name: String!
    url: String!
    description: String
    created: DateTime!
}
```

Здесь мы создали собственный скалярный тип — `DateTime`. Теперь мы можем узнать, когда была создана (`created`) каждая фотография. Любое поле, отмеченное `DateTime`, вернет строку JSON, но мы можем применять собственный скаляр, чтобы убедиться, что строка может быть сериализована, проверена и отформатирована как официальные дата и время.

Вы можете объявлять пользовательские скаляры для любого типа, который требуется проверить.

В npm-пакете graphql-custom-types есть некоторые широко применяемые пользовательские скалярные типы, которые вы можете быстро добавить в свой сервис Node.js GraphQL.

Перечисления

Типы *перечислений*, или `enum`, являются скалярными типами, которые позволяют полю возвращать ограниченный набор строковых значений. Если вы хотите, чтобы поле возвращало одно значение из ограниченного набора значений, вы можете использовать тип `enum`.

Например, создадим тип `enum` с именем `PhotoCategory`, который определяет тип фотографии одним из пяти возможных вариантов — `SELFIE`, `PORTRAIT`, `ACTION`, `LANDSCAPE` или `GRAPHIC`:

```
enum PhotoCategory {
    SELFIE
    PORTRAIT
    ACTION
    LANDSCAPE
    GRAPHIC
}
```

При определении полей вы можете использовать типы перечисления. Добавим поле `category` к нашему типу объекта `Photo`:

```
type Photo {
    id: ID!
    name: String!
    url: String!
    description: String
    created: DateTime!
    category: PhotoCategory!
}
```

Теперь, когда мы добавили категорию, мы убедимся, что она вернет одно из пяти допустимых значений при реализации сервиса.

Неважно, имеет ли ваша реализация полную поддержку типов перечислений. Вы можете реализовать поля перечисления GraphQL на любом языке.

Соединения и списки

Когда вы создаете схемы GraphQL, вы можете определять поля, которые возвращают списки любого типа GraphQL. Списки создаются путем окружения типа GraphQL квадратными скобками. `[String]` определяет список строк, а `[PhotoCategory]` — список категорий фотографий. Как обсуждается в главе 3, списки также могут состоять

из нескольких типов, если мы используем типы `union` или `interface`. Мы более подробно обсудим эти типы списков ближе к концу данной главы.

Иногда восклицательный знак может приводить к сложностям при определении списков. Если восклицательный знак указывается после закрывающей квадратной скобки, это означает, что само поле не может быть нулевым. Если восклицательный знак указывается перед закрывающей квадратной скобкой, значения, содержащиеся в списке, не могут быть нулевыми. Везде, где вы видите восклицательный знак, требуется значение и нельзя вернуть `null`. В табл. 4.1 перечислены указанные ситуации.

Таблица 4.1. Правила нулевых значений со списками

Объявление списка	**Определение**
[Int]	Список целых значений, которые могут быть нулевыми
[Int!]	Список целых значений, которые не могут быть нулевыми
[Int]!	Ненулевой список целых значений, которые могут быть нулевыми
[Int!]!	Ненулевой список целых значений, которые не могут быть нулевыми

Большинство определений списков — ненулевые списки с ненулевыми значениями. Это происходит потому, что обычно не требуется, чтобы значения в списке были нулевыми. Мы должны отфильтровать любые нулевые значения ранее. Если наш список не содержит никаких значений, мы можем просто вернуть пустой массив JSON; например `[]`. Пустой массив технически не нулевой: это просто массив, который не содержит никаких значений.

Возможность подключения данных и запроса нескольких типов связанных данных — очень важная функция. Когда мы создаем списки наших пользовательских типов объектов, мы применяем эту мощную функцию и соединяем объекты друг с другом.

В этом разделе мы расскажем о том, как задействовать список для подключения типов объектов.

Соединения «один к одному»

Когда мы создаем поля на основе пользовательских типов объектов, мы соединяем два объекта. В теории графов соединение или связь между двумя объектами называется *ребром*. Первый тип соединения — это соединение «один к одному», в котором мы соединяем один тип объекта с одним типом другого объекта.

Фотографии публикуются пользователями, поэтому каждая фотография в нашей системе должна содержать ребро, соединяющее ее с пользователем, который ее разместил. На рис. 4.1 показана односторонняя связь между двумя типами: `Photo` и `User`. Ребро, соединяющее два узла, называется `postedBy`.

Рис. 4.1. Соединение «один к одному»

Посмотрим, как мы будем определять это в схеме:

```
type User {
    githubLogin: ID!
    name: String
    avatar: String
}

type Photo {
    id: ID!
    name: String!
    url: String!
    description: String
    created: DateTime!
    category: PhotoCategory!
    postedBy: User!
}
```

Мы добавили новый тип в схему `User`. Пользователи приложения PhotoShare собираются авторизовываться на сайте GitHub с по-

мощью аккаунта. Когда пользователь авторизуется, мы получаем его значение `githubLogin` и применяем как уникальный идентификатор для своей записи пользователя. При желании, если пользователь добавит свое имя или фотографию в GitHub, мы сохраним эту информацию в полях `name` и `avatar`.

Затем мы добавили соединение, включив поле `postedBy` в объект фотографии. Каждая фотография должна быть опубликована пользователем, так что для этого поля установлен тип `User!`; восклицательный знак добавляется, чтобы сделать указанное поле ненулевым.

Соединения «один ко многим»

Данная идея подходит для того, чтобы при возможности поддерживать сервисы GraphQL ненаправленными. Это дает нашим клиентам максимальную гибкость для создания запросов, поскольку они могут начать перемещение по графу с любого узла. Все, что нам нужно сделать, чтобы следовать указанной практике, — обеспечить путь назад от типов `User` к типам `Photo`. Это означает, что, когда мы запрашиваем `User`, мы должны увидеть все фотографии, которые опубликовал конкретный пользователь:

```
type User {
    githubLogin: ID!
    name: String
    avatar: String
    postedPhotos: [Photo!]!
}
```

Добавив поле `postedPhotos` к типу `User`, мы предоставили путь к `Photo` от пользователя. Поле `postedPhotos` будет возвращать список типов `Photo` — это фотографии, отправленные родительским пользователем. Поскольку один пользователь может опубликовать много фотографий, мы создали соединение «один ко многим». Соединения «один ко многим», как показано на рис. 4.2, являются общими, они создаются, когда родительский объект содержит поле, в котором перечислены другие объекты.

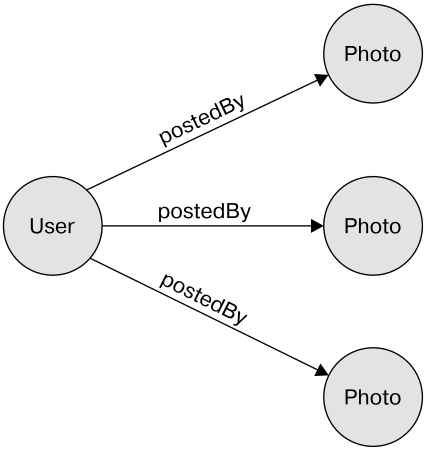

Рис. 4.2. Соединение «один ко многим»

Обычная позиция для добавления соединений «один ко многим» — это корневые типы. Чтобы наши фотографии или пользователи были доступны в запросе, нам нужно определить поля корневого типа `Query`. Посмотрим, как мы можем добавить новые пользовательские типы в корневой тип `Query`:

```
type Query {
    totalPhotos: Int!
    allPhotos: [Photo!]!
    totalUsers: Int!
    allUsers: [User!]!
}

schema {
    query: Query
}
```

Добавление типа `Query` определяет запросы, доступные в нашем API. В этом примере мы добавили два запроса для каждого типа: один — для доставки общего количества записей, доступных для каждого типа, а другой — для предоставления полного списка таких записей. Кроме того, мы добавили тип запроса `Query` в `schema` как файл. Это делает наши запросы доступными в нашем API GraphQL.

Теперь наши фотографии и пользователи могут быть выбраны со следующей строкой запроса:

```
query {
    totalPhotos
    allPhotos {
        name
        url
    }
}
```

Соединения «многие ко многим»

Иногда нам нужно соединить списки узлов с другими списками узлов. Наше приложение PhotoShare позволит пользователям идентифицировать других пользователей на каждой фотографии, которую они публикуют. Этот процесс называется *тегированием*. На фотографии может быть изображено несколько пользователей, и каждый из них может быть отмечен на нескольких фотографиях, как показано на рис. 4.3.

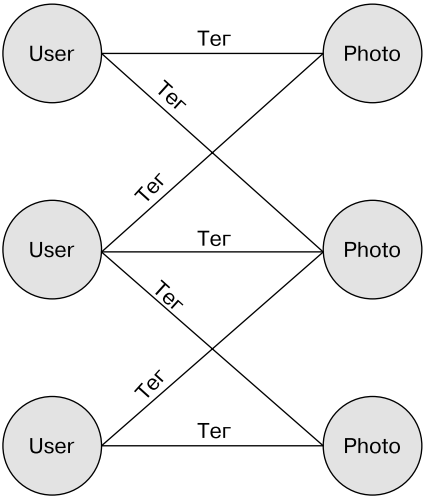

Рис. 4.3. Соединение «многие ко многим»

Чтобы создать такой тип соединения, нам нужно добавить поля списка как к типам `User`, так и к типам `Photo`:

```
type User {
    ...
    inPhotos: [Photo!]!
}

type Photo {
    ...
    taggedUsers: [User!]!
}
```

Как вы можете видеть, соединение *«многие ко многим»* состоит из двух соединений *«один ко многим»*. В этом случае на `Photo` может быть отмечено много пользователей и `User` может быть отмечен на многих фотографиях.

Сквозные типы. Иногда при создании отношений «многие ко многим» вам может потребоваться сохранить некоторую информацию о самом отношении. Поскольку в нашем приложении для обмена фотографиями нет реальной потребности в сквозном типе, мы собираемся применить другой пример для определения сквозного типа — дружбу между пользователями.

Мы можем подключить многих пользователей друг к другу, указав ниже `User` поле, которое содержит список других пользователей:

```
type User {
    friends: [User!]!
}
```

Здесь мы определяем список друзей для каждого пользователя. Рассмотрим случай, когда нам нужно сохранить некоторую информацию о самой дружбе, например о том, как давно пользователи знают друг друга или где они встретились.

В этой ситуации нам нужно определить ребро как тип пользовательского объекта. Мы называем данный объект *сквозным типом*, потому что это узел, который предназначен для соединения двух узлов. Определим сквозной тип `Friendship`, который мы можем применять для подключения двух друзей, а также предоставления данных о том, как связаны друзья:

```
type User {
    friends: [Friendship!]!
}
type Friendship {
    friend_a: User!
    friend_b: User!
    howLong: Int!
    whereWeMet: Location
}
```

Вместо определения поля `friends` непосредственно в списке других типов `User` мы создали `Friendship` для подключения к `friends`. Тип `Friendship` определяет двух подключенных друзей: `friend_a` и `friend_b`. Он также определяет некоторые поля с подробностями о том, как связаны друзья: `howLong` и `whereWeMet`. Поле `howLong` — это `Int`, который определит время дружбы, а поле `whereWeMet` связывается с пользовательским типом, называемым `Location`.

Мы можем улучшить дизайн типа `Friendship`, позволяя группе друзей быть частью дружбы. Например, возможно, вы встретили своих лучших друзей в одно и то же время в первом классе. Мы можем позволить двум и более друзьям стать частью дружбы, добавив одно поле `friends`:

```
type Friendship {
    friends: [User!]!
    how_long: Int!
    where_we_met: Location
}
```

Мы только включили одно поле для всех `friends` в `Friendship`. Теперь этот тип может отражать двух и более друзей.

Списки разных типов

В GraphQL списки не всегда должны возвращать один и тот же тип. В главе 3 мы вводили типы `union` и `interfaces`, и вы узнали, как писать запросы для этих типов с применением фрагментов. Посмотрим, как мы можем добавить указанные типы в нашу схему.

Здесь мы будем использовать расписание в качестве примера. У вас может быть расписание, состоящее из разных событий, каждое из

которых требует разные поля данных. Например, информация о семинаре или отработке занятия может быть совершенно иной, но вы должны иметь возможность добавить оба расписания. Вы можете думать о ежедневном расписании как о списке различных видов деятельности.

Есть два способа, которыми мы можем справиться с определением схемы для графика в GraphQL: объединения и интерфейсы.

Типы объединения

В GraphQL *тип объединения* — это тип, который мы можем использовать для возврата одного из нескольких разных типов. Вспомните из главы 3, как мы написали запрос, называемый `schedule`, который задавал повестку дня и возвращал разные данные, когда в качестве пункта повестки дня была отработка занятия или семинар. Еще раз взглянем на код:

```
 query schedule {
     agenda {
         ...on Workout {
            name
            reps
         }
         ...on StudyGroup {
            name
            subject
            students
         }
    }
}
```

Для ежедневного расписания дня ученика мы также могли бы использовать тип объединения, называемый `AgendaItem`:

```
union AgendaItem = StudyGroup | Workout

type StudyGroup {
    name: String!
    subject: String
    students: [User!]!
}

type Workout {
    name: String!
```

```
    reps: Int!
}

type Query {
    agenda: [AgendaItem!]!
}
```

`AgendaItem` объединяет семинары и отработки занятий в виде одного типа. Когда мы добавляем поле `agenda` в наш запрос, мы определяем его как список отработок или семинаров.

Можно объединить столько типов, сколько нужно в рамках единого объединения. Просто отделите каждый тип вертикальной чертой:

```
union = StudyGroup | Workout | Class | Meal | Meeting | FreeTime
```

Интерфейсы

Другим способом обработки полей, которые могут содержать разные типы, являются интерфейсы. *Интерфейсы* представляют собой абстрактные типы, которые могут быть реализованы как типы объекта. Интерфейс определяет все поля, которые должны быть включены в любой объект, который его реализует. Интерфейсы — отличный способ организовать код в вашей схеме. Это гарантирует, что определенные типы всегда включают определенные поля, которые запрашиваются независимо от возвращаемого типа.

В главе 3 мы написали запрос для повестки дня (`agenda`), в котором использовался интерфейс для возврата полей по разным элементам в расписании. Рассмотрим его здесь:

```
query schedule {
  agenda {
    name
    start
    end
    ...on Workout {
      reps
    }
  }
}
```

Вот как это могло бы выглядеть для запроса `agenda`, в котором реализованы интерфейсы. Для типа интерфейса с нашим расписанием он

должен содержать определенные поля, которые будут реализованы во всех пунктах повестки дня. Данные поля включают `name`, а также `start` и `end` (для времени). Неважно, какой тип элемента расписания у нас есть, все они нуждаются в уточнении, чтобы быть включенными в расписание.

Вот как мы могли бы реализовать такое решение в нашей схеме GraphQL:

```
scalar DataTime

interface AgendaItem {
    name: String!
    start: DateTime!
    end: DateTime!
}

type StudyGroup implements AgendaItem {
    name: String!
    start: DateTime!
    end: DateTime!
    participants: [User!]!
    topic: String!
}

type Workout implements AgendaItem {
    name: String!
    start: DateTime!
    end: DateTime!
    reps: Int!
}

type Query {
    agenda: [AgendaItem!]!
}
```

В этом примере мы создаем интерфейс `AgendaItem`. Он является абстрактным типом, который могут реализовать другие типы. Когда другой тип реализует интерфейс, он должен содержать поля, определенные интерфейсом. Оба типа, `StudyGroup` и `Workout`, реализуют интерфейс `AgendaItem`, поэтому им нужно использовать поля имени, начала и конца. В списке `agenda` запроса отображается пере-

чень типов `AgendaItem`. Любой тип, который реализует интерфейс `AgendaItem`, может быть возвращен в списке `agenda`.

Обратите внимание и на то, что эти типы также могут реализовывать другие поля. В `StudyGroup` есть `topic` и список участников, и у отработки занятия все еще есть повторы. Вы можете выбрать данные дополнительные поля в запросе с помощью фрагментов.

Объединения и интерфейсы — это инструменты, которые можно использовать для создания полей, содержащих разные типы объектов. Вам решать, когда задействовать тот или иной. В общем случае, если объекты содержат совершенно разные поля, рекомендуется применять объединения. Они очень эффективны. Если тип объекта должен содержать определенные поля для взаимодействия с другим типом объекта, вам нужно будет использовать интерфейс, а не объединение.

Аргументы

Аргументы могут быть добавлены в любое поле в GraphQL. Они позволяют нам отправлять данные, которые могут повлиять на результаты наших операций в GraphQL. В главе 3 мы рассмотрели аргументы пользователя в наших запросах и мутациях. Теперь посмотрим, как мы будем определять аргументы в нашей схеме.

Тип `Query` содержит поля, которые будут перечислять `allUsers` или `allPhotos`, но что происходит, когда вы хотите выбрать только одного пользователя или одну фотографию? Вам нужно будет предоставить некоторую информацию об одном пользователе или выбранной фотографии. Вы можете отправить эту информацию вместе с моим запросом в качестве аргумента:

```
type Query {
    ...
    User(githubLogin: ID!): User!
    Photo(id: ID!): Photo!
}
```

Точно так же, как поле, аргумент должен иметь тип. Данный тип может быть определен с использованием любого из скалярных типов объектов, доступных в нашей схеме. Чтобы выбрать конкретного

пользователя, нам нужно отправить его уникальный `githubLogin` в качестве аргумента. Следующий запрос выбирает только имя и аватар `MoonTahoe`:

```
query {
    User(githubLogin: "MoonTahoe") {
        name
        avatar
    }
}
```

Чтобы выбрать информацию об отдельной фотографии, необходимо указать ее идентификатор:

```
query {
    Photo(id: "14TH5B6NS4KIG3H4S") {
        name
        description
        url
    }
}
```

В обоих случаях аргументы требовались для запроса сведений об одной конкретной записи. Поскольку эти аргументы обязательны, они определяются как поля, не допускающие нулевое значение. Если мы не предоставляем `id` или `githubLogin` в этих запросах, парсер GraphQL вернет ошибку.

Фильтрация данных

Аргументы не должны быть ненулевыми. Мы можем добавить необязательные аргументы, используя нулевые поля. Это означает, что мы можем поставлять аргументы в качестве необязательных параметров при выполнении операций запроса. Например, отфильтровать список фотографий, возвращаемый запросом `allPhotos`, по категории фотографий:

```
type Query {
    ...
    allPhotos(category: PhotoCategory): [Photo!]!
}
```

Мы добавили необязательное поле `category` в запрос `allPhotos`. Категория должна соответствовать значениям типа перечисления `PhotoCategory`. Если значение не отправляется с запросом, можно предположить, что это поле вернет каждую фотографию. Однако, если категория предоставлена, мы должны получить отфильтрованный список фотографий в той же категории:

```
query {
    allPhotos(category: "SELFIE") {
        name
        description
        url
    }
}
```

Этот запрос возвращает `name`, `description` и `url` каждой фотографии, классифицированной как `SELFIE`.

Пагинация данных

Если наше приложение PhotoShare станет популярным, а оно станет, в нем будет много пользователей (`Users`) и фотографий (`Photos`). Возврат каждого `User` или каждого `Photo` в нашем приложении может оказаться невозможным. Мы можем применять аргументы GraphQL для управления объемом данных, возвращаемых из наших запросов. Этот процесс называется *разбиением данных на страницы* или *пагинацией*, потому что для представления одной страницы данных возвращается определенное количество записей.

Чтобы реализовать пагинацию данных, мы добавим два необязательных аргумента: `first`, чтобы собрать количество записей, которые должны быть возвращены сразу на одной странице данных, и `start`, чтобы определить начальную позицию или индекс первой записи для возврата. Мы можем добавить эти аргументы в оба наших списка запросов:

```
type Query {
    ...
    allUsers(first: Int=50 start: Int=0): [User!]!
    allPhotos(first: Int=25 start: Int=0): [Photo!]!
}
```

В предыдущем примере мы добавили необязательные аргументы `first` и `start`. Если клиент не предоставит эти аргументы в запросе, мы будем применять предоставленные по умолчанию значения. По умолчанию запрос `allUsers` возвращает только первых 50 пользователей, а запрос `allPhotos` возвращает только первые 25 фотографий.

Клиент может запросить другой диапазон пользователей или фотографий, указав значения этих аргументов. Например, если нам нужно выбрать пользователей с номерами от 90 до 100, мы могли бы сделать это, применяя следующий запрос:

```
query {
    allUsers(first: 10 start: 90) {
        name
        avatar
    }
}
```

Данный запрос выбирает только десять записей, начиная с 90-го пользователя. Он должен вернуть `name` и `avatar` для указанного диапазона пользователей. Мы можем вычислить общее количество страниц, доступных на клиенте, разделив общее количество элементов на размер одной страницы данных:

```
pages = pageSize/total
```

Сортировка

При запросе списка данных мы также можем определить способ сортировки возвращаемого списка данных. Для этого также применяются аргументы.

Рассмотрим сценарий, в котором мы хотим включить возможность сортировки любых списков записей `Photo`. Один из способов решения указанной задачи — создание перечислений, которые определяют, какие поля можно использовать для сортировки объектов `Photo`, и добавление инструкции для сортировки этих полей:

```
enum SortDirection {
    ASCENDING
```

```
    DESCENDING
}

enum SortablePhotoField {
    name
    description
    category
    created
}

Query {
    allPhotos(
        sort: SortDirection = DESCENDING
        sortBy: SortablePhotoField = created
    ): [Photo!]!
}
```

Здесь мы добавили аргументы `sort` и `sortBy` в запрос `allPhotos`. Мы создали тип перечисления, называемый `SortDirection`, который мы можем использовать, чтобы ограничить значения аргумента `sort` как `ASCENDING` или `DESCENDING`. Мы также создали другой тип перечисления, `SortablePhotoField`. Мы не хотим сортировать фотографии по любому полю, поэтому ограничили значения `sortBy`, чтобы включить только четыре поля фотографии: `name`, `description`, `category` или `created` (дата и время добавления фотографии). И `sort`, и `sortBy` являются необязательными аргументами, так что по умолчанию применяются значения `DESCENDING` и `created` соответственно, если какой-либо из аргументов не указан.

Теперь клиенты могут управлять сортировкой фотографий при выдаче запроса `allPhotos`:

```
query {
    allPhotos(sortBy: name)
}
```

Этот запрос вернет все фотографии, отсортированные по имени в нисходящем порядке.

До сих пор мы добавляли аргументы только к полям типа `Query`, но важно отметить, что вы можете добавлять аргументы к любому полю. Мы могли бы добавить параметры фильтрации, сортировки

и пагинации к фотографиям, которые были отправлены одним пользователем:

```
type User {
    postedPhotos(
        first: Int = 25
        start: Int = 0
        sort: SortDirection = DESCENDING
        sortBy: SortablePhotoField = created
        category: PhotoCategory
    ): [Photo!]
```

Добавление фильтров пагинации может помочь уменьшить объем данных, которые возвращает запрос. Мы обсудим идею ограничения данных более подробно в главе 7.

Мутации

Мутации должны определяться в схеме. Подобно запросам, они также определяются в своем собственном пользовательском типе объекта и добавляются в схему. Технически нет никакой разницы между тем, как мутация или запрос определены в вашей схеме. Разница в намерении. Мы должны создавать мутации только тогда, когда действие или событие что-то изменит в отношении состояния нашего приложения.

Мутации можно представить как *глаголы* в вашем приложении. Они должны состоять из того, что пользователи могут *делать* с вашим сервисом. При разработке сервиса GraphQL создайте список всех действий, которые пользователь может выполнить с вашим приложением. Скорее всего, это и будут ваши мутации.

В приложении PhotoShare пользователи могут войти в систему с помощью GitHub, опубликовать и сопроводить тегами фотографии. Все эти действия меняют что-то в состоянии приложения. После авторизации через аккаунт GitHub обращения текущих пользователей к клиенту будут меняться. Когда пользователь отправляет фотографию, в системе будет добавлено дополнительное фото. То же самое верно для добавления тегов к фотографии. Новые

записи данных тегов генерируются каждый раз при добавлении тега к фотографии.

Мы можем добавить эти мутации в корневой тип мутации в нашей схеме и сделать их доступными для клиента. Начнем с нашей первой мутации, `postPhoto`:

```
type Mutation {
    postPhoto(
        name: String!
        description: String
        category: PhotoCategory=PORTRAIT
    ): Photo!
}

schema {
    query: Query
    mutation: Mutation
}
```

Добавление поля `postPhoto` типа `Mutation` позволяет пользователям публиковать фотографии. Или, по крайней мере, публиковать метаданные о фотографиях. Мы рассмотрим загрузку фактических фотографий в главе 7.

Когда пользователь отправляет фотографию, для нее требуется как минимум `name`. `description` и `category` являются необязательными. Если аргумент `category` не указан, опубликованная фотография будет установлена по умолчанию как `PORTRAIT`. Например, пользователь может опубликовать фотографию, отправив следующую мутацию:

```
mutation {
    postPhoto(name: "Sending the Palisades") {
        id
        url
        created
        postedBy {
            name
        }
    }
}
```

После того как пользователь отправляет фотографию, он может выбирать информацию о фотографии, которую только что опубликовал. Это хорошо, потому что некоторые данные о новой фотографии будут сгенерированы на сервере. В базе данных будет создан `ID` новой фотографии. Будет автоматически сгенерирован ее `url`. На фото также будут отмечены дата и время создания (`created`) фотографии. Данный запрос выбирает все эти новые поля после публикации фотографии.

Кроме того, выборка включает информацию о пользователе, который опубликовал фотографию. Пользователь должен авторизоваться, чтобы опубликовать фотографию. Если в настоящий момент он не авторизован, эта мутация должна возвращать ошибку. Предполагая, что пользователь авторизован, мы можем получить информацию о том, кто опубликовал фотографию, с помощью поля `postedBy`. В главе 5 мы рассмотрим, как аутентифицировать авторизованного пользователя, применяя токен доступа.

Переменные мутации

Когда вы используете мутации, полезно объявлять для них переменные, как вы это делали в главе 3. Это делает вашу мутацию многоразовой при создании многих пользователей, а также подготавливает вас к применению этой мутации для реального клиента. Для краткости мы опустили этот шаг для остальной части главы, но вот как это выглядит:

```
mutation postPhoto(
    $name: String!
    $description: String
    $category: PhotoCategory
) {
    postPhoto(
        name: $name
        description: $description
        category: $category
    ) {
        id
        name
        email
    }
}
```

Типы ввода

Как вы могли заметить, аргументы для нескольких наших запросов и мутаций становятся довольно длинными. Существует лучший способ организовать эти аргументы с использованием *типов ввода*. Тип ввода аналогичен типу объекта GraphQL, за исключением того, что применяется только для входных аргументов.

Улучшим мутацию `postPhoto` с использованием типа ввода для наших аргументов:

```
input PostPhotoInput {
  name: String!
  description: String
  category: PhotoCategory=PORTRAIT
}

type Mutation {
    postPhoto(input: PostPhotoInput!): Photo!
}
```

Тип `PostPhotoInput` похож на тип объекта, но он был создан только для входных аргументов. Для этого требуются поля `name` и `description`, а поля `category` по-прежнему являются необязательными. Теперь при отправке мутации `postPhoto` сведения о новой фотографии должны быть включены в один объект:

```
mutation newPhoto($input: PostPhotoInput!) {
    postPhoto(input: $input) {
        id
        url
        created
    }
}
```

Когда мы создаем эту мутацию, мы устанавливаем тип переменной запроса `$input` для соответствия нашему типу ввода `PostPhotoInput!`. Он не является нулевым, потому что как минимум нам нужно получить доступ к полю `input.name`, чтобы добавить новую фотографию. Когда мы отправляем мутацию, необходимо

предоставить новые фотоданные в наших переменных запроса, вложенных в поле `input`:

```
{
    "input": {
        "name": "Hanging at the Arc",
        "description": "Sunny on the deck of the Arc",
        "category": "LANDSCAPE"
    }
}
```

Наш ввод сгруппирован в объект JSON и отправлен вместе с мутацией в переменных запроса в виде ключа `"input"`. Поскольку переменные запроса отформатированы как JSON, категория должна быть строкой, которая соответствует одной из категорий типа `PhotoCategory`.

Типы ввода — это ключ к организации и написанию четкой схемы GraphQL. Вы можете добавлять типы ввода как аргументы в любом поле для улучшения пагинации данных и фильтрации данных в приложениях.

Посмотрим, как мы можем организовать и повторно использовать все поля сортировки и фильтрации с помощью типов ввода:

```
input PhotoFilter {
    category: PhotoCategory
    createdBetween: DateRange
    taggedUsers: [ID!]
    searchText: String
}

input DateRange {
    start: DateTime!
    end: DateTime!
}

input DataPage {
    first: Int = 25
    start: Int = 0
}

input DataSort {
```

```
    sort: SortDirection = DESCENDING
    sortBy: SortablePhotoField = created
}

type User {
    ...
    postedPhotos(filter:PhotoFilter paging:DataPage
                 sorting:DataSort): [Photo!]!
    inPhotos(filter:PhotoFilter paging:DataPage
             sorting:DataSort): [Photo!]!
}

type Photo {
    ...
    taggedUsers(sorting:DataSort): [User!]!
}

type Query {
    ...
    allUsers(paging:DataPage sorting:DataSort): [User!]!
    allPhotos(filter:PhotoFilter paging:DataPage
              sorting:DataSort): [Photo!]!
}
```

Мы организовали множество полей в виде типов ввода и повторно использовали эти поля в качестве аргументов в нашей схеме.

Типы ввода `PhotoFilter` содержат необязательные поля ввода, которые позволяют клиенту фильтровать список фотографий. Тип `PhotoFilter` включает вложенный тип ввода `DateRange` в виде поля `createdBetween`. `DateRange` должен включать даты начала и окончания. Используя `PhotoFilter`, мы также можем фильтровать фотографии по категориям, строке поиска или `taggedUsers`. Мы добавляем все параметры фильтра в каждое поле, которое возвращает список фотографий. Это дает клиенту большой контроль над тем, какие фотографии возвращаются из каждого списка.

Типы ввода также были созданы для пагинации и сортировки. Тип ввода `DataPage` содержит поля, необходимые для запроса страницы данных, а тип ввода `DataSort` — наши поля сортировки. Эти типы ввода были добавлены в каждое поле в нашей схеме, которое возвращает список данных.

Мы могли бы написать запрос, который принимает некоторые довольно сложные входные данные, используя доступные типы ввода:

```
query getPhotos($filter:PhotoFilter $page:DataPage
$sort:DataSort) {
    allPhotos(filter:$filter paging:$page sorting:$sort) {
        id
        name
        url
    }
}
```

Этот запрос не обязательно принимает аргументы для трех типов ввода: `$filter`, `$page` и `$sort`. Используя переменные запроса, мы можем отправить некоторые конкретные сведения о том, какие фотографии хотели бы выбрать:

```
{
    "filter": {
        "category": "ACTION",
        "taggedUsers": ["MoonTahoe", "EvePorcello"],
        "createdBetween": {
            "start": "2018-11-6",
            "end": "2018-5-31"
        }
    },
    "page": {
        "first": 100
    }
}
```

В этом запросе будут найдены все фотографии в категории `ACTION`, на которых GitHub-пользователи `MoonTahoe` и `EvePorcello` поставили теги с 6 ноября по 31 мая, то есть во время лыжного сезона. Кроме того, здесь мы запрашиваем первые 100 фотографий.

Типы ввода помогают упорядочивать схемы и использовать аргументы. Они также улучшают документацию схемы, которую автоматически генерирует GraphiQL или GraphQL Playground. Это сделает API более доступным и более простым в освоении и оценке. Наконец, вы можете применять типы ввода, чтобы предоставить клиенту большие возможности для выполнения организованных запросов.

Возвращаемые типы

Все поля в нашей схеме возвращают наши основные типы — `User` и `Photo`. Но иногда нам нужно возвращать метаинформацию о запросах и мутациях в дополнение к фактическим данным полезной нагрузки. Например, когда пользователь авторизовался и прошел аутентификацию, нам нужно вернуть токен в дополнение к полезной нагрузке для `User`.

Чтобы авторизоваться с помощью GitHub OAuth, мы должны получить код OAuth от GitHub. Мы обсудим настройку учетной записи GitHub OAuth и получение кода GitHub в разделе «Авторизация с помощью аккаунта GitHub» главы 5. Пока предположим, что у нас есть допустимый код GitHub, который мы можем отправить мутации `githubAuth` для авторизации пользователя:

```
type AuthPayload {
    user: User!
    token: String!
}

type Mutation {
    ...
    githubAuth(code: String!): AuthPayload!
}
```

Пользователи аутентифицируются путем отправки допустимого кода GitHub в мутацию `githubAuth`. В случае успеха мы вернем тип пользовательского объекта, содержащий как информацию об успешно авторизованном пользователе, так и токен, который может применяться для авторизации дополнительных запросов и мутаций, включая мутацию `postPhoto`.

Можно задействовать пользовательские типы возвращаемых данных в любом поле, для которого нужны более простые данные полезной нагрузки. Возможно, необходимо знать, сколько времени требуется запросу для доставки ответа или сколько результатов было найдено в конкретном ответе в дополнение к данным полезной нагрузки запроса. Вы можете обрабатывать все это с помощью пользовательского возвращаемого типа.

На данном этапе мы представили все типы, доступные вам при создании схем GraphQL. Мы даже потратили немного времени на обсуждение техник, которые помогут вам улучшить дизайн схемы. Но есть один последний тип корневого объекта, который нам нужно представить, — тип `Subscription`.

Подписки

Типы `Subscription` не отличаются от любого другого типа объекта на языке определения схемы GraphQL. Здесь мы определяем доступные подписки как поля в пользовательском типе объектов. Мы должны будем убедиться, что подписки реализуют шаблон проектирования Pub/Sub вместе с каким-либо видом транспорта реального времени, когда будем создавать сервис GraphQL позже в главе 7.

Например, мы можем добавить подписки, которые позволят нашим клиентам быть в курсе создания новых типов `Photo` или `User`:

```
type Subscription {
    newPhoto: Photo!
    newUser: User!
}
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

Здесь мы создаем пользовательский объект `Subscription`, который содержит два поля: `newPhoto` и `newUser`. Когда будет опубликовано новое фото, эта новая фотография будет передана всем клиентам, оформившим подписку `newPhoto`. Когда будет создан новый пользователь, его данные передадутся каждому клиенту, подписанному на оповещения о новых пользователях.

Подобно запросам или мутациям, подписки могут использовать аргументы. Предположим, нам нужно добавить фильтры к подписке `newPhoto`, которая будет оповещать только о новых фотографиях в категории `ACTION`:

```
type Subscription {
    newPhoto(category: PhotoCategory): Photo!
    newUser: User!
}
```

Когда пользователи оформляют подписку `newPhoto`, у них появляется возможность фильтровать фотографии, которые были добавлены к этой подписке. Например, чтобы фильтровать только новые фотографии в категории `ACTION`, клиенты могли бы отправить следующую операцию в наш API GraphQL:

```
subscription {
    newPhoto(category: "ACTION") {
        id
        name
        url
        postedBy {
            name
        }
    }
}
```

Такая подписка должна возвращать данные только для фотографий в категории `ACTION`.

Подписка — отличное решение, когда важно обрабатывать данные в реальном времени. В главе 7 мы больше поговорим о реализации подписок для всех ваших задач обработки данных в режиме реального времени.

Документация схемы

В главе 3 объясняется, как в GraphQL работает система самодиагностики, которая может информировать вас о том, какие запросы поддерживаются сервером. При написании схемы GraphQL вы можете добавить необязательные описания для каждого поля, которые предоставят дополнительную информацию о типах и полях схемы. Предоставление описаний может облегчить вашей команде, вам и другим пользователям API понимание вашей системы типов.

Например, добавим комментарии к типу `User` в нашей схеме:

```
"""
A user who has been authorized by GitHub at least once
"""
type User {

    """
    The user's unique GitHub login
    """
    githubLogin: ID!

    """
    The user's first and last name
    """
    name: String

    """
    A url for the user's GitHub profile photo
    """
    avatar: String

    """
    All of the photos posted by this user
    """
    postedPhotos: [Photo!]!

    """
    All of the photos in which this user appears
    """
    inPhotos: [Photo!]!

}
```

Добавив три кавычки выше и ниже вашего комментария к каждому типу или полю, вы предоставляете пользователям словарь для вашего API. Помимо типов и полей, вы также можете документировать аргументы. Взглянем на мутацию `postPhoto`:

```
Replace with:

  type Mutation {
```

```
    """
    Authorizes a GitHub User
    """
    githubAuth(
      "The unique code from GitHub that is sent to authorize
      the user"
      code: String!
    ): AuthPayload!
}
```

Комментарии аргументов предоставляют имя аргумента и информацию о том, не является ли поле необязательным. Если вы используете типы ввода, вы можете документировать их, как и любой другой тип:

```
"""
The inputs sent with the postPhoto Mutation
"""
input PostPhotoInput {
  "The name of the new photo"
  name: String!
  "(optional) A brief description of the photo"
  description: String
  "(optional) The category that defines the photo"
  category: PhotoCategory=PORTRAIT
}

postPhoto(
      "input: The name, description, and category for a new photo"
      input: PostPhotoInput!
): Photo!
```

Все эти комментарии затем перечисляются в документации схемы GraphQL Playground или GraphiQL, как показано на рис. 4.4. Конечно, вы также можете задать интроспективные запросы, чтобы найти описания данных типов.

В основе всех проектов GraphQL лежит прочная, четко определенная схема. И это служит дорожной картой и контрактом между фронтенд- и бэкенд-командами для обеспечения того, чтобы собранный продукт всегда соответствовал схеме.

```
postPhoto(
  input: PostPhotoInput!
): Photo!

  Adds a new photo

  Arguments input: The name,
  description, and category for a new
  photo
```

Рис. 4.4. Документация postPhoto

В этой главе мы создали схему для нашего фотоприложения. В следующих трех главах мы покажем вам, как создать полноценное приложение GraphQL на основе только что созданной схемы.

5 API GraphQL

Вы узнали историю GraphQL. Вы написали несколько запросов. Вы создали схему. Теперь вы готовы создать полностью функционирующий сервис GraphQL. Это можно реализовать с помощью ряда различных технологий, но мы будем использовать JavaScript. Методы, которые здесь показаны, в основном универсальны, поэтому, даже если в деталях реализации различаются, общая архитектура будет схожей независимо от того, какой язык или фреймворк вы выберете.

Если вас интересуют серверные библиотеки для других языков, вы можете найти многие из них на сайте GraphQL.org.

Спецификация GraphQL, выпущенная в 2015 году, была сосредоточена на ясном объяснении языка запросов и системы типов. В ней намеренно не указывалась подробная информация о реализации сервера, чтобы разработчики на разных языках могли использовать то, что им более удобно. Команда в Facebook действительно предоставила реализацию, которую они разработали на языке JavaScript и назвали GraphQL.js. Наряду с этим они выпустили модуль *express-graphql*, предоставляющий простой способ создания сервера GraphQL с Express, и, в частности, первую библиотеку, чтобы помочь разработчикам выполнить указанную задачу.

После изучения реализаций JavaScript на серверах GraphQL мы решили использовать Apollo Server, проект с открытым исходным кодом от команды Apollo. Apollo Server довольно прост в настройке и предлагает множество готовых функций, включая поддержку подписок, загрузку файлов, API источника данных для быстрого

подключения существующих сервисов и интеграцию с Apollo Engine из коробки. Он также включает в себя платформу GraphQL Playground для написания запросов непосредственно в браузере.

Настройка проекта

Начнем с создания проекта `photo-share-api` в виде пустой папки на компьютере. Помните: вы всегда можете посетить репозиторий Learning GraphQL, чтобы увидеть завершенный проект или посмотреть, как работает проект на Glitch. Внутри этой папки мы создадим новый проект npm, выполнив команду `npm init -y` в оболочке командной строки. Данная команда создаст файл `package.json` и установит все параметры по умолчанию, так как мы использовали флаг `-y`.

Затем мы установим зависимости проекта: `apollo-server` и `graphql`. Мы также установим `nodemon`:

```
npm install apollo-server graphql nodemon
```

`apollo-server` и `graphql` необходимы для установки экземпляра Apollo Server. `nodemon` будет сканировать файлы на предмет изменений и перезапускать сервер при их внесении. Таким образом, нам не придется останавливать и перезапускать сервер каждый раз, когда мы вносим изменения. Добавим команду для `nodemon` в файл `package.json` на ключ `scripts`:

```
"scripts": {
  "start": "nodemon -e js,json,graphql"
}
```

Теперь каждый раз, когда мы запускаем `npm start`, выполняется файл `index.js`, и `nodemon` будет следить за изменениями в любых файлах с расширением `js`, `json` или `graphql`. Кроме того, нам нужно создать файл `index.js` в корне проекта. Убедитесь, что в файле `package.json` в качестве основного файла указан документ `index.js`:

```
"main": "index.js"
```

Распознаватели

В нашей книге мы много раз фокусировались на запросах. Схема определяет операции запроса, разрешенные клиентам, а также то, как связаны разные типы. Схема описывает требования к данным, но не выполняет работу по получению этих данных. Такая работа обрабатывается распознавателями.

Распознаватель — это функция, которая возвращает данные для определенного поля. Функции `Resolver` возвращают данные в типе и форме, заданных схемой. Распознаватели могут быть асинхронными и могут извлекать или обновлять данные из REST API, базы данных или любого другого сервиса.

Посмотрим, как может выглядеть распознаватель для нашего корневого запроса. В файле `index.js` в корне проекта добавим поле `totalPhotos` в `Query`:

```
const typeDefs = `
    type Query {
        totalPhotos: Int!
    }
`


const resolvers = {
  Query: {
    totalPhotos: () => 42
  }
}
```

Переменная `typeDefs` определяет нашу схему. Это просто строка. Всякий раз, когда мы создаем такой запрос, как `totalPhotos`, он должен быть подкреплен функцией распознавателя с тем же именем. Определение типа описывает тип, который должно возвращать поле. Функция-распознаватель возвращает данные этого типа — в нашем случае просто статичное значение `42`.

Также важно отметить, что распознаватель должен быть определен как объект с тем же именем, что и объект в схеме. Поле `totalPhotos` является частью объекта запроса. Распознаватель для этого поля также должен быть частью объекта `Query`.

Мы создали начальные определения типов для нашего корневого запроса. Мы также создали наш первый распознаватель, который поддерживает поле запроса `totalPhotos`. Чтобы создать схему и запустить выполнение запросов к ней, мы будем использовать Apollo Server:

```
// 1. Требуется 'apollo-server'.
const { ApolloServer } = require('apollo-server')

const typeDefs = `
        type Query {
                totalPhotos: Int!
        }
`

const resolvers = {
  Query: {
    totalPhotos: () => 42
  }
}

// 2. Создаем новый экземпляр сервера.
// 3. Отправляем ему объект с typeDefs (схема) и resolvers.
const server = new ApolloServer({
  typeDefs,
  resolvers
})

// 4. Вызываем отслеживание на сервере для запуска веб-сервера.
server
  .listen()
  .then(({url}) => console.log(`GraphQL Service running
     on ${url}`))
```

После запроса ApolloServer создадим новый экземпляр сервера, отправив ему объект с двумя значениями: `typeDefs` и `resolvers`. Это быстрая и минимальная настройка сервера, которая позволяет нам поддерживать мощный API GraphQL. Позже в данной главе мы поговорим о том, как расширить функциональность сервера с помощью Express.

На этом этапе мы готовы выполнить запрос для `totalPhotos`. Когда мы выполним команду `npm start`, то увидим, что GraphQL

Playgroud запущен по адресу `http://localhost:4000`. Попробуем следующий запрос:

```
{
    totalPhotos
}
```

Возвращенное значение `totalPhotos` равно `42`:

```
{
  "data": {
    "totalPhotos": 42
  }
}
```

Распознаватели являются ключевыми инструментами для реализации GraphQL. Каждое поле должно иметь соответствующую функцию распознавателя. Распознаватель должен следовать правилам схемы — иметь то же имя, что и поле, которое было определено в схеме, и возвращать тип данных, заданный схемой.

Корневые распознаватели

Как обсуждалось в главе 4, API GraphQL имеют корневые типы `Query`, `Mutation` и `Subscription`. Эти типы находятся на верхнем уровне и представляют собой все возможные точки входа в API. Ранее мы добавили поле `totalPhotos` к типу `Query`, и это означает, что наш API может запросить данное поле.

Внесем изменения, создав корневой тип `Mutation`. Поле мутации называется `postPhoto` и будет принимать строковые аргументы `name` и `description`. Когда мутация отправляется, она должна возвращать логическое значение (`Boolean`):

```
const typeDefs = `
    type Query {
        totalPhotos: Int!
    }

    type Mutation {
    postPhoto(name: String! description: String): Boolean!
  }
`
```

После создания мутации `postPhoto` нужно добавить соответствующий распознаватель в объект `resolvers`:

```
// 1. Тип данных для хранения ваших фотографий в памяти.
var photos = []

const resolvers = {
  Query: {

    // 2. Возвращаем длину массива фотографий.
    totalPhotos: () => photos.length

  },

  // 3. Распознаватель Mutation и postPhoto.
  Mutation: {
    postPhoto(parent, args) {
        photos.push(args)
        return true
    }
  }
}
```

В первую очередь следует создать переменную `photos` для хранения сведений о фотографии в массиве. Позже в этой главе мы будем хранить фотографии в базе данных.

Затем мы расширяем распознавание `totalPhotos` так, чтобы вернуть длину массива фотографий. Всякий раз, когда поле запрашивается, оно возвращает количество фотографий, которые в настоящее время хранятся в массиве.

Здесь добавляется распознаватель `postPhoto`. На этот раз мы используем аргументы с нашей функцией `postPhoto`. Первый аргумент — ссылка на родительский объект. В документации такие ссылки могут быть представлены как `_`, `root` или `obj`. В нашем случае родительский объект распознавателя `postPhoto` является мутацией. Родитель в настоящее время не содержит каких-либо данных, которые нам нужны, но это всегда первый аргумент, передаваемый распознавателю. Поэтому нам нужно добавить аргумент `parent`, чтобы

мы могли получить доступ ко второму аргументу, передаваемому распознавателю: аргументы мутации.

Второй аргумент, передаваемый распознавателю `postPhoto`, — это аргументы GraphQL, которые были отправлены в данную операцию: `name` и (не обязательно) `description`. Переменная `args` — объект, который содержит эти два поля: `{name, description}`. Сейчас аргументы характеризуют один объект фотографии, поэтому мы вводим их непосредственно в массив `photos`.

Пришло время протестировать мутацию `postPhoto` на платформе GraphQL Playground, отправив строку для аргумента `name`:

```
mutation newPhoto {
    postPhoto(name: "sample photo")
}
```

Данная мутация добавляет сведения о фотографии в массив и возвращает `true`. Модифицируем эту мутацию для использования переменных запроса:

```
mutation newPhoto($name: String!, $description: String) {
    postPhoto(name: $name, description: $description)
}
```

После добавления переменных в мутацию данные должны передаваться для предоставления строковых переменных. В нижнем левом углу платформы Playground добавим значения для имени и описания в окне Query Variables (Переменные запроса):

```
{
    "name": "sample photo A",
    "description": "A sample photo for our dataset"
}
```

Распознаватели типов

Когда выполняется GraphQL-запрос, мутация или подписка, возвращается результат по форме запроса. Мы видели, что распознаватели могут возвращать значения скалярного типа, такие как целые числа, строки и логические элементы, но они также могут возвращать и объекты.

Для нашего фотоприложения мы создадим тип `Photo` и поле запроса `allPhotos`, которое вернет список объектов `Photo`:

```
const typeDefs = `

  # 1. Добавляем определение типа Photo.
    type Photo {
      id: ID!
      url: String!
      name: String!
      description: String
    }

  # 2. Возвращаем Photo по запросу allPhotos.
    type Query {
      totalPhotos: Int!
      allPhotos: [Photo!]!
    }

  # 3. Возвращаем недавно опубликованную фотографию из мутации.
    type Mutation {
      postPhoto(name: String! description: String): Photo!
    }
```

Поскольку мы добавили объект `Photo` и запрос `allPhotos` к нашим определениям типов, нам необходимо отразить эти изменения в распознавателях. Мутация `postPhoto` должна возвращать данные в форме типа `Photo`. Запрос `allPhotos` должен вернуть список объектов, которые имеют ту же форму, что и тип `Photo`:

```
// 1. Переменная, которую мы будем увеличивать
// для уникальных идентификаторов.
var _id = 0

var photos = []

const resolvers = {
  Query: {
    totalPhotos: () => photos.length,
    allPhotos: () => photos
  },
  Mutation: {
```

```
    postPhoto(parent, args) {

      // 2. Создаем новую фотографию и генерируем идентификатор.
        var newPhoto = {
          id: _id++,
          ...args
      }
      photos.push(newPhoto)

      // 3. Возвращаем новую фотографию.
      return newPhoto
    }
  }
}
```

Поскольку для типа `Photo` требуется идентификатор, мы создали для его хранения переменную. В распознавателе `postPhoto` мы будем генерировать идентификаторы, увеличивая это значение. Переменная `args` предоставляет поля имени и описания для фотографии, но нам также нужен идентификатор. Обычно сервер создает переменные, такие как идентификаторы и метки времени. Таким образом, когда мы создаем новый объект фотографии в распознавателе `postPhoto`, мы добавляем поле `ID` и распространяем поля `name` и `description` из `args` в наш новый объект фотографии.

Вместо того чтобы возвращать логическое значение, мутация возвращает объект, который соответствует типу `Photo`. Этот объект создается сгенерированным идентификатором и полями `name` и `description`, которые были переданы с данными. Кроме того, мутация `postPhoto` добавляет фотообъекты в массив `photos`. Эти объекты соответствуют типу `Photo`, который мы определили в нашей схеме, поэтому мы можем вернуть весь массив фотографий на запрос `allPhotos`.

Создание уникальных идентификаторов с инкрементируемой переменной, безусловно, является немасштабируемым способом создания идентификаторов, но будет служить нашим целям здесь в качестве демонстрации. В реальном приложении ваши идентификаторы, скорее всего, будут генерироваться базой данных.

Чтобы убедиться, что `postPhoto` работает правильно, мы можем настроить мутацию. Поскольку `Photo` является типом, нам нужно добавить выборку к нашей мутации:

```
mutation newPhoto($name: String!, $description: String) {
    postPhoto(name: $name, description: $description) {
        id
        name
        description
    }
}
```

После добавления нескольких фотографий с помощью мутаций следующий запрос `allPhotos` должен вернуть массив всех добавленных объектов `Photo`:

```
query listPhotos {
    allPhotos {
        id
        name
        description
    }
}
```

Мы также добавили в нашу схему фотографий поля с ненулевыми `url`. Что произойдет, если мы добавим поле `url` в выборку?

```
query listPhotos {
    allPhotos {
        id
        name
        description
        url
    }
}
```

Когда поле `url` добавлено в выборку запроса, отображается сообщение об ошибке: Не удается вернуть значение null для поля Photo.url с ненулевым значением. Мы не будем добавлять поле `url` в выборку, так как нам не нужно сохранять URL-адреса, потому что они могут генерироваться автоматически. Каждое поле в нашей схеме может отображаться на распознаватель. Все, что нам нужно сделать, — это добавить объект `Photo` в список распознавателей и определить поля,

которые необходимо сопоставить с функциями. В данном случае мы бы использовали функцию, помогающую распознать URL-адреса:

```
const resolvers = {
  Query: { ... },
  Mutation: { ... },
  Photo: {
    url: parent => `http://yoursite.com/img/${parent.id}.jpg`
  }
}
```

Поскольку мы будем применять распознаватель для URL-адресов фотографий, мы добавили объект `Photo` в распознаватели. Распознаватель `Photo`, добавленный к корню, называется *тривиальным распознавателем* (trivial resolver). Тривиальные распознаватели добавляются на верхний уровень объекта `resolvers`, но они необязательны. У нас есть возможность создавать пользовательские распознаватели для объекта `Photo`, применяя тривиальный распознаватель. Если вы не укажете тривиальный распознаватель, GraphQL вернется к распознавателю по умолчанию, который возвращает свойство с тем же именем, что и поле.

Когда мы выбираем поле `url` фотографии в нашем запросе, вызывается соответствующая функция распознавания. Первый аргумент, посланный распознавателям, всегда является родительским объектом `parent`. В этом случае объект `parent` представляет текущий объект `Photo`, который распознается. Мы предполагаем, что наш сервис обрабатывает только изображения в формате JPEG. Данные изображения именуются согласно их идентификатору и располагаются по адресу http://yoursite.com/img/. Поскольку `parent` описывает фотографию, мы можем получить идентификатор фотографии с помощью этого аргумента и использовать его для автоматического создания URL-адреса текущего снимка.

Когда мы определяем схему GraphQL, мы описываем требования к данным нашего приложения. С помощью распознавателей можно эффективно и гибко выполнять эти требования. Функции предоставляют нам такую силу и гибкость. Функции могут быть асинхронными, возвращать скалярные типы, объекты и данные из разных источников. Распознаватели — это просто функции, и каждое поле в нашей схеме GraphQL может отображаться в распознавателе.

Использование вводов и перечислений

Пришло время ввести тип перечисления `PhotoCategory` и тип ввода `PostPhotoInput` для нашей схемы `typeDefs`:

```
enum PhotoCategory {
  SELFIE
  PORTRAIT
  ACTION
  LANDSCAPE
  GRAPHIC
}

type Photo {
  ...
  category: PhotoCategory!
}

input PostPhotoInput {
  name: String!
  category: PhotoCategory=PORTRAIT
  description: String
}

type Mutation {
  postPhoto(input: PostPhotoInput!): Photo!
}
```

В главе 4 мы создали эти типы, когда разрабатывали схему для приложения `PhotoShare`. Мы также добавили тип перечисления `PhotoCategory` и добавили поле `category` к нашим фотографиям. При разрешении фотографий мы должны убедиться, что категория фотографий — строка, которая соответствует значениям, указанным в типе перечисления, — доступна. Мы также должны собрать категорию, когда пользователи публикуют новые фотографии.

Мы добавили тип `PostPhotoInput` для организации аргумента для мутации `postPhoto` в одном объекте. Этот тип ввода имеет поле категории. Даже если пользователь не предоставляет поле категории в качестве аргумента, будет применяться значение по умолчанию `PORTRAIT`.

В код распознавателя `postPhoto` нам нужно внести некоторые изменения. Сведения о фотографии, `name`, `description` и `category` теперь вложены в поле `input`. Нам нужно убедиться, что мы обращаемся к этим значениям в `args.input` вместо `args`:

```
postPhoto(parent, args) {
    var newPhoto = {
        id: _id++,
        ...args.input
    }
    photos.push(newPhoto)
    return newPhoto
}
```

Теперь мы запускаем мутацию с новым типом ввода:

```
mutation newPhoto($input: PostPhotoInput!) {
  postPhoto(input:$input) {
    id
    name
    url
    description
    category
  }
}
```

Нам также нужно отправить соответствующий JSON-код в окне Query Variables (Переменные запроса):

```
{
  "input": {
    "name": "sample photo A",
    "description": "A sample photo for our dataset"
  }
}
```

Если категория не указана, по умолчанию будет установлено значение `PORTRAIT`. В качестве альтернативы, если для категории указано значение, оно будет проверено на соответствие типу перечисления до того, как операция будет даже отправлена на сервер. Если это допустимая категория, она будет передана распознавателю в качестве аргумента.

С типами ввода мы можем сделать передачу аргументов для мутаций более универсальной и менее подверженной ошибкам. При объединении типов ввода и перечислений мы можем конкретизировать типы входов, которые могут быть предоставлены для определенных полей. Входы и перечисления невероятно ценны, но еще лучше использовать их вместе.

Ребра и соединения

Как мы уже говорили ранее, мощь GraphQL заключается в ребрах — соединениях между узлами данных. Когда вы запускаете сервер GraphQL, типы обычно сопоставляются с моделями. Эти типы можно представить как сохраняемые в таблицах схожие данные. Аналогично мы связываем типы с соединениями. Рассмотрим виды соединений, которые мы можем применять для определения взаимосвязанных отношений между типами.

Соединения «один ко многим»

Пользователям необходимо получить доступ к списку фотографий, которые они ранее разместили. Мы получим доступ к этим данным в поле `postedPhotos`, превращающемся в фильтрованный список фотографий, которые опубликовал пользователь. Поскольку один `User` может публиковать много `Photos`, мы называем это *отношением «один ко многим»*. Добавим `User` в нашу схему `typeDefs`:

```
type User {
  githubLogin: ID!
  name: String
  avatar: String
  postedPhotos: [Photo!]!
}
```

На данном этапе мы создали ориентированный граф. Мы можем перейти от типа `User` к типу `Photo`. Чтобы получить неориентированный граф, нам нужно обеспечить обратный путь к типу `User` от типа `Photo`. Добавим поле `postedBy` в тип `Photo`:

```
type Photo {
  id: ID!
  url: String!
```

```
  name: String!
  description: String
  category: PhotoCategory!
  postedBy: User!
}
```

Добавив поле `postedBy`, мы создали ссылку на `User`, который разместил `Photo`, создав неориентированный граф. Это соединение «один к одному», потому что одна фотография может быть отправлена только одним `User`.

Пример пользователей

Чтобы протестировать наш сервер, добавим некоторые данные для примера в наш файл `index.js`. Обязательно удалите имеющуюся переменную `photos`, которой присвоен пустой массив:

```
var users = [
  { "githubLogin": "mHattrup", "name": "Mike Hattrup" },
  { "githubLogin": "gPlake", "name": "Glen Plake" },
  { "githubLogin": "sSchmidt", "name": "Scot Schmidt" }
]

var photos = [
  {
    "id": "1",
    "name": "Dropping the Heart Chute",
    "description": "The heart chute is one of my favorite chutes",
    "category": "ACTION",
    "githubUser": "gPlake"
  },
  {
    "id": "2",
    "name": "Enjoying the sunshine",
    "category": "SELFIE",
    "githubUser": "sSchmidt"
  },
  {
    id: "3",
    "name": "Gunbarrel 25",
    "description": "25 laps on gunbarrel today",
    "category": "LANDSCAPE",
    "githubUser": "sSchmidt"
  }
]
```

Поскольку соединения создаются с применением полей объекта, их можно сопоставлять с функциями распознавателя. Внутри этих функций мы можем использовать сведения о родителе, чтобы помочь найти и вернуть связанные данные.

Добавим в наш код распознаватели `postedPhotos` и `postedBy`:

```
const resolvers = {
  ...
  Photo: {
    url: parent => `http://yoursite.com/img/${parent.id}.jpg`,
    postedBy: parent => {
      return users.find(u => u.githubLogin === parent.githubUser)
    }
  },
  User: {
    postedPhotos: parent => {
      return photos.filter(p => p.githubUser ===
                           parent.githubLogin)
    }
}
```

В распознавателе `Photo` нам нужно добавить поле для `postedBy`. Внутри этого распознавателя требуется определить, как найти связанные данные. Задействуя метод массива `.find()`, мы можем получить пользователя, для которого `githubLogin` соответствует значению `githubUser`, сохраненному с каждой фотографией. Метод `.find()` возвращает один объект пользователя.

В распознавателе `User` мы извлекаем список фотографий, отправленных этим пользователем, задействуя метод `.filter()` массива. Данный метод возвращает массив только тех фотографий, которые содержат значение `githubUser`, соответствующее родительскому значению `githubLogin` пользователя. Метод фильтра возвращает массив фотографий.

Теперь попробуем отправить запрос `allPhotos`:

```
query photos {
  allPhotos {
    name
    url
    postedBy {
```

```
      name
    }
  }
}
```

Когда мы запрашиваем каждую фотографию, мы можем запросить сведения о пользователе, разместившем эту фотографию. Пользовательский объект обнаруживается и возвращается распознавателем. В указанном примере мы выбираем только имя пользователя, разместившего фотографию. Учитывая наши данные в примере, результат должен вернуть следующий JSON-код:

```
{
  "data": {
    "allPhotos": [
      {
        "name": "Dropping the Heart Chute",
        "url": "http://yoursite.com/img/1.jpg",
        "postedBy": {
          "name": "Glen Plake"
        }
      },
      {
        "name": "Enjoying the sunshine",
        "url": "http://yoursite.com/img/2.jpg",
        "postedBy": {
          "name": "Scot Schmidt"
        }
      },
      {
        "name": "Gunbarrel 25",
        "url": "http://yoursite.com/img/3.jpg",
        "postedBy": {
          "name": "Scot Schmidt"
        }
      }
    ]
  }
}
```

Мы отвечаем за соединение данных с помощью распознавателей, но, как только мы сможем вернуть связанные данные, наши

клиенты смогут отправлять сложные запросы. В следующем разделе мы покажем вам некоторые методы создания соединений «многие ко многим».

Соединения «многие ко многим»

Следующее, что нам нужно добавить в наше приложение, — возможность отмечать пользователей на фотографиях. Это означает, что `User` может быть помечен на многих разных фотографиях, а у `Photo` может быть много разных пользователей, отмеченных на нем. Отношения, которые создаются фототегами между пользователями и фотографиями, можно назвать *«многие ко многим»* — многие пользователи для многих фотографий.

Чтобы упростить отношения «многие ко многим», мы добавляем поле `taggedUsers` в `Photo` и поле `inPhotos` в `User`. Изменим код схемы `typeDefs`:

```
type User {
      ...
      inPhotos: [Photo!]!
  }

  type Photo {
      ...
      taggedUsers: [User!]!
}
```

Поле `taggedUsers` возвращает список пользователей, а поле `inPhotos` возвращает список фотографий, в которых отображается пользователь. Чтобы упростить эту связь «многие ко многим», нам нужно добавить массив тегов. Чтобы проверить функцию тегирования, вам нужно заполнить некоторые примеры данных для тегов:

```
var tags = [
    { "photoID": "1", "userID": "gPlake" },
    { "photoID": "2", "userID": "sSchmidt" },
    { "photoID": "2", "userID": "mHattrup" },
    { "photoID": "2", "userID": "gPlake" }
]
```

Когда у нас есть фотография, мы должны произвести поиск по нашим выборкам, чтобы найти пользователей, отмеченных на фото-

графии. Когда у нас есть пользователь, мы можем найти список фотографий, в которых он появляется. Поскольку наши данные в настоящее время хранятся в массивах JavaScript, мы будем применять методы массива внутри распознавателей, чтобы найти данные:

```
Photo: {
    ...
    taggedUsers: parent => tags

      // Возвращает массив тегов, которые содержат только
      // текущую фотографию
      .filter(tag => tag.photoID === parent.id)

      // Преобразует массив тегов в массив значений userID
      .map(tag => tag.userID)

      // Преобразует массив значений userID в массив объектов
      // пользователей
      .map(userID => users.find(u => u.githubLogin === userID))

},
User: {
  ...
  inPhotos: parent => tags

    // Возвращает массив тегов, которые содержат
    // только текущего пользователя
    .filter(tag => tag.userID === parent.id)

    // Преобразует массив тегов в массив значений photoID
    .map(tag => tag.photoID)

    // Преобразует массив значений photoID в массив объектов
    // фотографий
    .map(photoID => photos.find(p => p.id === photoID))

}
```

Распознаватель поля `taggedUsers` отфильтровывает любые фотографии, которые не являются текущей, и отображает этот отфильтрованный список как массив фактических объектов `User`. Распознаватель поля `inPhotos` фильтрует теги пользователей и сопоставляет теги пользователя с массивом фактических объектов `Photo`.

Теперь мы можем просмотреть, какие пользователи отмечены на каждой фотографии, отправив GraphQL-запрос:

```
query listPhotos {
  allPhotos {
    url
    taggedUsers {
      name
    }
  }
}
```

Возможно, вы заметили, что у нас есть массив для тегов, но у нас нет типа GraphQL под названием `Tag`. GraphQL не требует, чтобы наши модели данных точно соответствовали типам в нашей схеме. Наши клиенты могут находить отмеченных пользователей на каждой фотографии и фотографиях, на которых отмечены те или иные пользователи, запрашивая тип `User` или `Photo`. Им не нужно запрашивать тип `Tag`: это лишь усложнит ситуацию. Мы уже преодолели тяжелый путь поиска помеченных пользователей или фотографий в нашем распознавателе специально для того, чтобы клиенты могли запросить указанные данные.

Пользовательские скаляры

Как обсуждалось в главе 4, GraphQL содержит набор стандартных скалярных типов по умолчанию, которые можно применять для любых полей. Скаляры, такие как `Int`, `Float`, `String`, `Boolean` и `ID`, подходят для большинства ситуаций, но могут быть случаи, когда нужно создать собственный скалярный тип в соответствии с вашими требованиями к данным.

Когда нужно реализовать пользовательский скаляр, необходимо создать правила о том, как тип должен быть сериализован и проверен. Например, если мы создадим тип `DateTime`, нам нужно будет определить, что должно считаться допустимым значением `DateTime`.

Добавим этот пользовательский скаляр `DateTime` к нашим `typeDefs` и применим его в типе `Photo` для поля `created`. Поле `created` используется для хранения даты и времени, когда была опубликована определенная фотография:

```
const typeDefs = `
  scalar DateTime
  type Photo {
    ...
    created: DateTime!
  }
  ...
`
```

Каждое поле в нашей схеме должно сопоставляться с распознавателем. Поле `created` необходимо сопоставить с распознавателем типа `DateTime`. Мы создали пользовательский скалярный тип для `DateTime`, потому что хотим анализировать и проверять любые поля, которые применяют этот скаляр как типы `Date` JavaScript.

Рассмотрим различные способы представления даты и времени в виде строки. Все эти строки представляют собой допустимые даты:

- `"4/18/2018"`;
- `"4/18/2018 1:30:00 PM"`;
- `"Sun Apr 15 2018 12:10:17 GMT-0700 (PDT)"`;
- `"2018-04-15T19:09:57.308Z"`.

Мы можем использовать любую из этих строк для создания объектов `datetime` с помощью JavaScript-кода:

```
var d = new Date("4/18/2018")
console.log( d.toISOString() )
// "2018-04-18T07:00:00.000Z"
```

Здесь мы создали новый объект даты, задействуя один формат, а затем преобразовали эту строку `datetime` в строку даты в формате ISO.

Все, что объект `Date` JavaScript не поддерживает, является недопустимым. Вы можете попытаться проанализировать следующие данные:

```
var d = new Date("Tuesday March")
console.log( d.toString() )
// Недопустимая дата
```

Когда мы запрашиваем поле `created` фотографии, нужно убедиться, что значение, возвращаемое этим полем, содержит строку в формате даты и времени ISO. Всякий раз, когда поле возвращает

значение даты, мы сериализуем (`serialize`) это значение как строку в формате ISO:

```
const serialize = value => new Date(value).toISOString()
```

Функция `serialize` получает значения поля из нашего объекта, и пока это поле содержит дату, отформатированную как объект JavaScript, или любую допустимую строку `datetime`, всегда будет возвращаться GraphQL в формате ISO `datetime`.

Когда ваша схема реализует собственный скаляр, он может быть использован в качестве аргумента в запросе. Предположим, что мы создали фильтр для запроса `allPhotos`. Этот запрос вернет список фотографий, сделанных после определенной даты:

```
type Query {
  ...
  allPhotos(after: DateTime): [Photo!]!
}
```

Если бы у нас было такое поле, клиенты могли бы отправить нам запрос, содержащий значение `DateTime`:

```
query recentPhotos(after:DateTime) {
  allPhotos(after: $after) {
    name
    url
  }
}
```

И они могли бы отправлять аргумент `$after`, используя переменные запроса:

```
{
  "after": "4/18/2018"
}
```

Нам нужно убедиться, что аргумент `after` разобран в JavaScript-объект `Date`, до его отправки в распознаватель:

```
const parseValue = value => new Date(value)
```

Мы можем применять функцию `parseValue` для анализа значений входящих строк, которые отправляются вместе с запросами. Возвращаемое значение `parseValue` передается аргументам распознавателя:

```
const resolvers = {
  Query: {
    allPhotos: (parent, args) => {
      args.after // Объект данных JavaScript
      ...
    }
  }
}
```

Пользовательские скаляры должны иметь возможность сериализовать и анализировать значения даты. Есть еще кое-что, что нам нужно для обработки строк даты. Это ситуация, когда клиенты добавляют строку даты непосредственно к самому запросу:

```
query {
  allPhotos(after: "4/18/2018") {
    name
    url
  }
}
```

Аргумент `after` не передается как переменная запроса. Вместо этого он добавляется непосредственно в документ запроса. Прежде чем мы сможем разобрать данное значение, нам нужно получить его из запроса после того, как оно было преобразовано в абстрактное синтаксическое дерево (АСД). Мы используем функцию `parseLiteral` для получения нужных значений из документа запроса перед их анализом:

```
const parseLiteral = ast => ast.value
```

Функция `parseLiteral` применяется для получения значения даты, которое было добавлено непосредственно в документ запроса. В таком случае все, что нам нужно сделать, — вернуть это значение, но при необходимости мы могли бы выполнить дополнительные шаги для разбора внутри этой функции.

Нам нужны все три функции, которые мы разработали для обработки значений `DateTime` при создании собственного скаляра. Добавим в наш код распознаватель для нашего пользовательского скаляра `DateTime`:

```
const { GraphQLScalarType } = require('graphql')
...
const resolvers = {
```

```
  Query: { ... },
  Mutation: { ... },
  Photo: { ... },
  User: { ... },
  DateTime: new GraphQLScalarType({
      name: 'DateTime',
      description: 'A valid date time value.',
      parseValue: value => new Date(value),
      serialize: value => new Date(value).toISOString(),
      parseLiteral: ast => ast.value
  })
}
```

Мы применяем объект `GraphQLScalarType` для создания распознавателей для пользовательских скаляров. Распознаватель `DateTime` помещается в наш список распознавателей. При создании нового скалярного типа нужно добавить три функции: `serialize`, `parseValue` и `parseLiteral`, которые будут обрабатывать любые поля или аргументы, реализующие скаляр `DateType`.

Примеры дат

Внутри данных также обязательно добавим ключ `created` и значение даты для двух существующих фотографий. Любая допустимая строка даты или объект даты будут обработаны, потому что поле `created` будет сериализовано до его возврата:

```
var photos = [
  {
    ...
    "created": "3-28-1977"
  },
  {
    ...
    "created": "1-2-1985"
  },
  {
    ...
    "created": "2018-04-15T19:09:57.308Z"
  }
]
```

Теперь, когда мы добавляем поля `DateTime` к выборкам, мы можем видеть эти даты и типы, отформатированные как строки в формате даты ISO:

```
query listPhotos {
  allPhotos {
    name
    created
  }
}
```

Осталось только научиться добавлять временную метку к каждой фотографии при отправке. Мы достигаем этого путем добавления поля `created` к каждой фотографии и отметки времени с помощью текущего значения `DateTime`, используя JavaScript-объект `Date`:

```
postPhoto(parent, args) {
    var newPhoto = {
        id: _id++,
        ...args.input,
        created: new Date()
    }
    photos.push(newPhoto)
    return newPhoto
}
```

Теперь при публикации новых фотографий они будут помечены датой и временем их создания.

Сервер apollo-server-express

В некоторых ситуациях вам может понадобиться добавить Apollo Server в существующее приложение, или, возможно, вы захотите воспользоваться промежуточным программным обеспечением Express. В этом случае вы можете задействовать `apollo-server-express`. С помощью Apollo Server Express вы сможете применять все новейшие функции Apollo Server, а также реализовать собственную конфигурацию. Для наших целей мы собираемся реорганизовать сервер для использования Apollo Server Express, чтобы настроить собственный домашний маршрут, маршрут Playground, и позднее разрешить загрузку и сохранение присланных изображений на сервере.

Начнем с удаления `apollo-server`:

```
npm remove apollo-server
```

Теперь установим Apollo Server Express и Express:

```
npm install apollo-server-express express
```

Express

Express на сегодняшний день является одним из самых популярных проектов в экосистеме Node.js. Он позволяет быстро и эффективно настраивать веб-приложения Node.js.

Теперь мы можем реорганизовать код в файле `index.js`. Мы начнем с изменения инструкции `require`, чтобы включить `apollo-server-express`. Затем мы добавим `express`:

```
// 1. Требуем `apollo-server-express` and `express`.
const { ApolloServer } = require('apollo-server-express')
const express = require('express')

...

// 2. Вызываем `express()`, чтобы создать приложение Express.
var app = express()

const server = new ApolloServer({ typeDefs, resolvers })

// 3. Вызываем `applyMiddleware()`, чтобы разрешить
// промежуточное ПО, смонтированное по тому же самому пути.
server.applyMiddleware({ app })

// 4. Создаем домашний маршрут.
app.get('/', (req, res) => res.end('Welcome to the PhotoShare
        API'))

// 5. Перехватываем события на определенном порте.
app.listen({ port: 4000 }, () =>
  console.log(`GraphQL Server running @ http://
  localhost:4000${server.graphqlPath}`)
)
```

Добавив Express, мы можем воспользоваться всеми функциями промежуточного программного обеспечения, предоставляемыми нам фреймворком. Чтобы ввести их на сервер, нам просто нужно вызвать функцию `express`, вызвать `applyMiddleware`, а затем настроить собственный маршрут. Теперь, когда мы перейдем по адресу http://localhost:4000, мы должны увидеть страницу с приветствием «Добро пожаловать в API PhotoShare». Пока это шаблон.

Затем нужно настроить собственный маршрут для GraphQL Playground для работы по адресу http://localhost:4000/playground. Мы можем сделать это, установив вспомогательный пакет с помощью менеджера npm. В первую очередь потребуется установить пакет `graphql-playground-middleware-express`:

```
npm install graphql-playground-middleware-express
```

Теперь укажем его в верхней части индексного файла:

```
const expressPlayground = require('graphql-playground-middleware-
express').default

...

app.get('/playground', expressPlayground({ endpoint:
'/graphql' }))
```

Затем мы будем использовать Express для создания маршрута для Playground, поэтому при необходимости запуска Playground мы укажем адрес http://localhost:4000/playground.

Теперь наш сервер настроен на запуск Apollo Server Express и у нас есть три различных маршрута:

- `/` — для домашней страницы;
- `/graphql` — для конечной точки GraphQL;
- `/playground` — для GraphQL Playground.

На данном этапе мы также уменьшим объем нашего индексного файла, переместив код `typeDefs` и `resolvers` в отдельные файлы.

Сначала мы создадим файл с именем `typeDefs.graphql` и поместим его в корень проекта. Это будет просто схема, только текст. Вы также можете переместить распознаватели в отдельную папку

resolvers. Вы можете разместить данные функции в файле index.js или сделать модульными файлы распознавателя так же, как мы делаем это в репозитории.

После завершения вы можете импортировать typeDefs и resolvers, как показано ниже. Мы будем использовать модуль fs из Node.js для чтения файла typeDefs.graphql:

```
const { ApolloServer } = require('apollo-server-express')
const express = require('express')
const expressPlayground =
  require('graphql-playground-middleware-express').default
const { readFileSync } = require('fs')
const typeDefs = readFileSync('./typeDefs.graphql', 'UTF-8')
const resolvers = require('./resolvers')

var app = express()

const server = new ApolloServer({ typeDefs, resolvers })

server.applyMiddleware({ app })

app.get('/', (req, res) => res.end('Welcome to
  the PhotoShare API'))
app.get('/playground', expressPlayground({ endpoint:
  '/graphql' }))

app.listen({ port: 4000 }, () =>
  console.log(`GraphQL Server running at
      http://localhost:4000${server.graphqlPath}`)
)
```

Теперь, реорганизовав сервер, мы готовы сделать следующий шаг — приступить к интеграции базы данных.

Контекст

В данном разделе мы рассмотрим *контекст*, в котором вы можете хранить глобальные значения. К ним может обратиться любой распознаватель. Контекст — хорошее место для хранения информации аутентификации, сведений о базе данных, локальных кэшей данных

и всего остального, что необходимо для распознавания операций GraphQL.

Вы можете напрямую обращаться к API REST и базам данных в своих распознавателях, но мы обычно абстрагируем эту логику в объект, который помещаем в контекст, чтобы обеспечить разделение проблем и позже предоставить более простую возможность рефакторизации. Вы также можете использовать контекст для доступа к данным REST из источника данных Apollo. Для получения дополнительных сведений ознакомьтесь с информацией об источниках данных Apollo в документации по адресу http://bit.ly/2vac9ZC.

Для наших целей, однако, мы собираемся включить контекст сейчас, чтобы устранить некоторые из существующих ограничений нашего приложения. Прежде всего мы сохраняем данные в памяти, что является не очень масштабируемым решением. Мы также небрежно обрабатываем идентификаторы, увеличивая их значения с каждой мутацией. Вместо этого мы будем полагаться на базу данных для поддержки хранения данных и генерации идентификаторов. Наши распознаватели смогут получить доступ к указанной базе данных из контекста.

Установка MongoDB

Для GraphQL неважно, какую базу данных вы применяете. Вы можете использовать Postgres, Mongo, SQL Server, Firebase, MySQL, Redis, Elastic — все, что хотите. Из-за популярности в сообществе Node.js мы будем применять Mongo в качестве решения для хранения данных для нашего приложения.

Чтобы начать работу с MongoDB на Mac, мы воспользуемся инструментом Homebrew. Чтобы установить Homebrew, посетите страницу https://brew.sh/. После того как вы установили Homebrew, установите Mongo, выполнив следующие команды:

```
brew install mongo
brew services list
brew services start
```

После успешного запуска MongoDB можно начать чтение и запись данных в локальный экземпляр Mongo.

Замечание для пользователей Windows

Если вы хотите запустить локальную версию MongoDB на Windows, перейдите по адресу http://bit.ly/inst-mdb-windows.

Вы также можете воспользоваться онлайн-сервисом Mongo, например mLab, как показано на рис. 5.1. Можете бесплатно создать базу данных в песочнице.

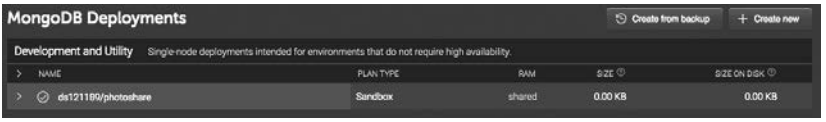

Рис. 5.1. Сервис mLab

Добавление базы данных к контексту

Теперь пришло время подключиться к нашей базе данных и соединиться с контекстом. Мы собираемся использовать пакет `mongodb` для связи с нашей базой данных. Мы можем установить его с помощью команды `npm install mongodb`.

После установки этого пакета мы изменим конфигурационный файл сервера Apollo, `index.js`. Придется подождать, пока `mongodb` не соединится с нашей базой данных, чтобы запустить сервис. Нам также нужно будет получить информацию о хосте базы данных из переменной среды `DB_HOST`. Мы сделаем эту переменную среды доступной в файле с расширением `.env` в корне проекта.

Если вы задействуете Mongo локально, URL-адрес будет выглядеть примерно так:

```
DB_HOST=mongodb://localhost:27017/<имя-вашей-базы-данных>
```

Если вы применяете mLab, URL-адрес будет иным. Обязательно создайте имя пользователя и пароль для базы данных и замените заполнители `<пользовательБД>` и `<парольБД>` своими значениями.

```
DB_HOST=mongodb://<пользовательБД>:<парольБД>@5555.mlab.com:
5555/<имя-вашей-базы-данных>
```

Подключимся к базе данных и создадим объект контекста перед запуском сервиса. Мы также будем применять пакет `dotenv` для загрузки URL-адреса `DB_HOST`:

```
const { MongoClient } = require('mongodb')
require('dotenv').config()

...

// 1. Создаем асинхронную функцию.
async function start() {
  const app = express()
  const MONGO_DB = process.env.DB_HOST

  const client = await MongoClient.connect(
    MONGO_DB,
    { useNewUrlParser: true }
  )
  const db = client.db()

  const context = { db }

  const server = new ApolloServer({ typeDefs, resolvers,
  context })

  server.applyMiddleware({ app })

  app.get('/', (req, res) => res.end('Welcome to the
  PhotoShare API'))

  app.get('/playground', expressPlayground({ endpoint:
  '/graphql' }))

  app.listen({ port: 4000 }, () =>
```

```
    console.log(
      `GraphQL Server running at
      http://localhost:4000${server.graphqlPath}`
    )
  )
}

// 5. Вызываем запуск при готовности.
start()
```

После запуска мы подключаемся к базе данных. Подключение к базе данных — асинхронный процесс. Для успешного подключения потребуется некоторое время. Эта асинхронная функция позволяет нам ожидать распознавания с помощью ключевого слова `await`. Первое, что мы делаем в данной функции, — ожидаем успешного подключения к локальной или удаленной базе данных. После подключения к базе данных мы можем добавить указанное соединение к объекту контекста и запустить наш сервер.

Теперь мы можем изменить наши распознаватели запросов, чтобы возвращать информацию из наших коллекций Mongo вместо локальных массивов. Мы также добавим запросы для `totalUsers` и `allUsers` и введем их в схему.

Схема

```
type Query {
    ...
    totalUsers: Int!
    allUsers: [User!]!
}
```

Распознаватели

```
Query: {

  totalPhotos: (parent, args, { db }) =>
      db.collection('photos')
        .estimatedDocumentCount(),

  allPhotos: (parent, args, { db }) =>
    db.collection('photos')
      .find()
```

```
      .toArray(),

  totalUsers: (parent, args, { db }) =>
    db.collection('users')
      .estimatedDocumentCount(),

  allUsers: (parent, args, { db }) =>
    db.collection('users')
      .find()
      .toArray()
}
```

Метод `db.collection('photos')` отвечает за получение доступа к коллекции Mongo. Мы можем посчитать документы в коллекции с помощью метода `.estimatedDocumentCount()`. Перечислить все документы в коллекции и преобразовать их в массив можно с помощью `.find()` и `.toArray()`. На данный момент коллекция `photos` пуста, но указанный код будет работать. Распознаватели `totalPhotos` и `totalUsers` ничего не должны возвращать. Распознаватели `allPhotos` и `allUsers` должны возвращать пустые массивы.

Чтобы добавить фотографии в базу данных, пользователь должен авторизоваться. В следующем разделе мы обрабатываем авторизацию пользователя с помощью GitHub и отправляем нашу первую фотографию в базу данных.

Авторизация с помощью аккаунта GitHub

Авторизация и аутентификация пользователей — важная часть любого приложения. Существует ряд стратегий, которые можно применить. Авторизация через социальные сети популярна, поскольку в этом случае много данных об управлении учетной записью передается из социальной сети. Пользователи могут чувствовать себя более защищенными при авторизации, поскольку социальная сеть им знакома и вызывает доверие. В приложении мы реализуем авторизацию через сайт GitHub, потому что очень вероятно, что у пользователя уже есть учетная запись GitHub (а если нет, то можно просто и быстро получить ее!)[1].

[1] Вы можете создать учетную запись на странице www.github.com.

Настройка GitHub OAuth

Прежде чем мы начнем, нужно настроить разрешения GitHub для работы данного приложения. Для этого выполните следующие действия.

1. Зайдите на сайт www.github.com и авторизуйтесь.
2. Выберите пункт меню Account Settings (Настройки аккаунта).
3. Перейдите в раздел Developer Settings (Настройки разработчика).
4. Щелкните кнопкой мыши на ссылке New OAuth App (Новое приложение авторизации).
5. Добавьте следующие настройки (рис. 5.2).
   - Application name (Название приложения) — Localhost 3000.
   - Homepage URL (URL-адрес домашней страницы) — http://localhost:3000.
   - Application description (Описание приложения) — All authorizations for local GitHub Testing.
   - Authorization callback URL (URL-адрес обратного вызова авторизации) — http://localhost:3000.

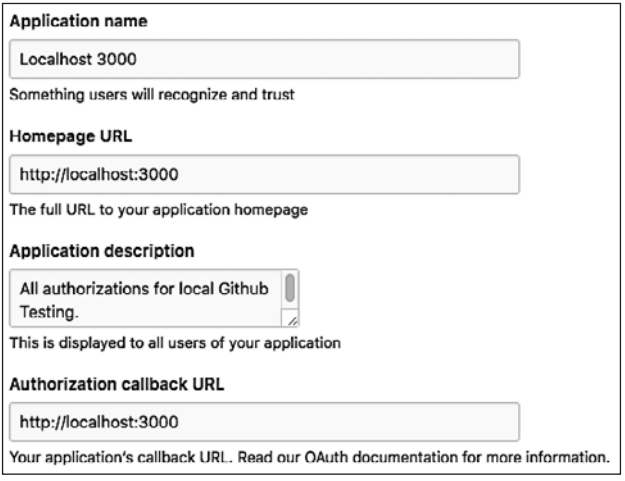

Рис. 5.2. Новое приложение OAuth

6. Нажмите кнопку Save (Сохранить).
7. Перейдите на страницу учетной записи OAuth и получите собственные значения `client_id` и `client_secret`, как показано на рис. 5.3.

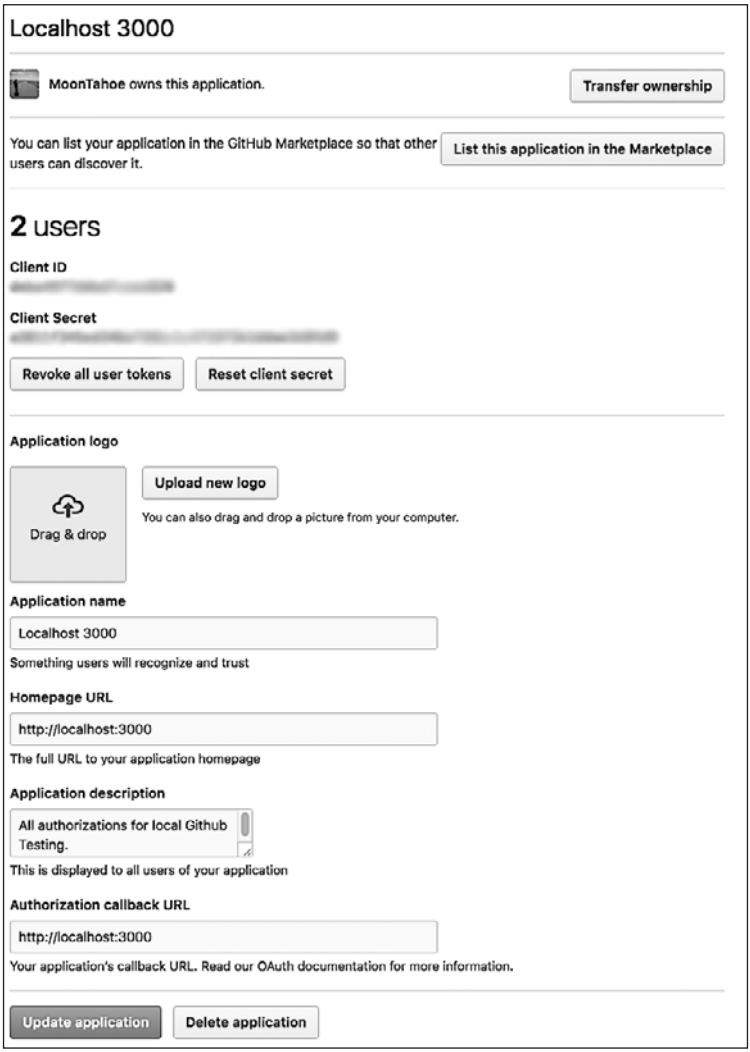

Рис. 5.3. Настройки приложения OAuth

С этой настройкой мы теперь можем получить токен аутентификации и информацию о пользователе от GitHub. В частности, нам понадобятся значения `client_id` и `client_secret`.

Процесс авторизации

Процесс авторизации приложения GitHub происходит на клиенте и на сервере. В данном разделе мы обсудим, как обращаться с сервером, а в главе 6 рассмотрим реализацию клиента. Как показано на рис. 5.4, полный процесс авторизации состоит из следующих этапов.

1. Клиент: запрашивает у GitHub код, задействуя URL-адрес с `client_id`.
2. Пользователь: получает доступ к информации учетной записи на GitHub для приложения клиента.
3. GitHub: отсылает код на URL-адрес перенаправления OAuth: http://localhost:3000?code=XYZ.
4. **Клиент: отсылает мутацию GraphQL `githubAuth(code)` с кодом.**

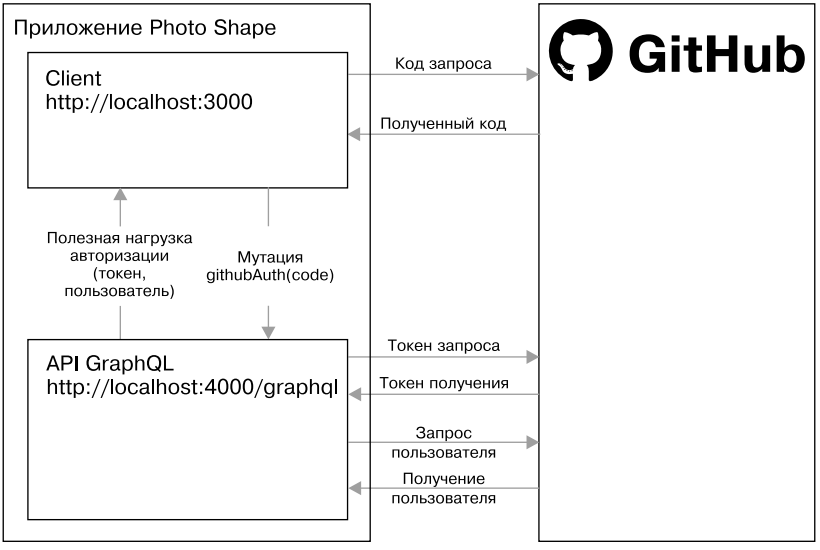

Рис. 5.4. Процесс авторизации

5. **API: запрашивает `access_token` GitHub с информацией: `client_id`, `client_secret` и `client_code`.**
6. **GitHub: возвращает `access_token`, который можно использовать в дальнейших запросах информации.**
7. **API: запрашивает информацию пользователя по `access_token`.**
8. **GitHub: отвечает информацией пользователя: `name`, `githubLogin` и `avatar`.**
9. **API: распознает мутацию `authUser(code)` с `AuthPayload`, где содержатся токен и пользователь.**
10. Клиент: сохраняет токен для пересылки с будущими запросами GraphQL.

Чтобы реализовать мутацию `githubAuth`, предположим, что у нас есть код. После того как мы применим код для получения токена, мы сохраним новую информацию пользователя и токен в нашей локальной базе данных и вернем эту информацию клиенту. Клиент будет сохранять токен локально и отправлять его нам с каждым запросом. Мы будем применять токен для авторизации пользователя и доступа к его записи в данных.

Мутация githubAuth

Мы обрабатываем авторизацию пользователей с помощью мутации GitHub. В главе 4 мы разработали пользовательский тип полезной нагрузки для нашей схемы, называемый `AuthPayload`. Добавим `AuthPayload` и мутацию `githubAuth` к нашим `typeDefs`:

```
type AuthPayload {
  token: String!
  user: User!
}

type Mutation {
  ...
  githubAuth(code: String!): AuthPayload!
}
```

Тип `AuthPayload` применяется только как ответ на мутации авторизации. Он содержит пользователя, авторизованного мутацией

вместе с токеном, который пользователь может задействовать для идентификации себя в будущих запросах.

Прежде чем мы запрограммируем распознаватель `githubAuth`, нам нужно создать две функции для обработки запросов API GitHub:

```
const requestGithubToken = credentials =>
    fetch(
        'https://github.com/login/oauth/access_token',
        {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
                Accept: 'application/json'
            },
            body: JSON.stringify(credentials)
        }
    )
    .then(res => res.json())
    .catch(error => {
      throw new Error(JSON.stringify(error))
    })
```

Функция `requestGithubToken` возвращает промис выборки. Учетные данные отправляются на URL API GitHub в теле запроса POST. Значение `credentials` состоит из трех элементов: `client_id`, `client_secret` и `code`. После выполнения функции ответ GitHub разбирается как JSON-код. Теперь мы можем использовать ее для запроса токена GitHub с `credentials`. Эту и будущие вспомогательные функции можно найти в файле `lib.js` в репозитории (https://github.com/MoonHighway/learning-graphql/blob/master/chapter-05/photo-share-api/lib.js/).

Как только у нас появится токен GitHub, нужно получить доступ к информации из учетной записи текущего пользователя. В частности, необходимы его GitHub-логин, имя и изображение профиля. Чтобы получить эту информацию, нужно отправить другой запрос в API GitHub вместе со значением `token`, которое мы получили из предыдущего запроса:

```
const requestGithubUserAccount = token =>
    fetch(`https://api.github.com/user?access_token=${token}`)
        .then(toJSON)
        .catch(throwError)
```

Данная функция также возвращает промис выборки. На этом маршруте API GitHub мы можем получить доступ к информации о текущем пользователе, если у нас есть токен доступа.

Теперь объединим оба данных запроса в одну асинхронную функцию, которую мы можем применять для авторизации пользователя с помощью GitHub:

```
async authorizeWithGithub(credentials) {
  const { access_token } = await requestGithubToken(credentials)
  const githubUser = await requestGithubUserAccount(access_token)
  return { ...githubUser, access_token }
}
```

Применение `async/await` здесь позволяет обрабатывать несколько асинхронных запросов. Сначала мы запрашиваем токен доступа и ожидаем ответа. Затем, задействуя `access_token`, запрашиваем информацию учетной записи пользователя GitHub и ожидаем ответа. После того как у нас будут данные, мы объединим все вместе в одном объекте.

Мы создали вспомогательные функции, которые будут поддерживать функциональность распознавателя. Теперь напишем распознаватель, чтобы получить токен и учетную запись пользователя из GitHub:

```
async githubAuth(parent, { code }, { db }) {
  // 1. Получаем данные от GitHub.
    let {
      message,
      access_token,
      avatar_url,
      login,
      name
    } = await authorizeWithGithub({
      client_id: <ID_ВАШЕГО_КЛИЕНТА>,
      client_secret: <ПАРОЛЬ_ВАШЕГО_КЛИЕНТА>,
      code
    })
  // 2. Если есть сообщение, что-то пошло не так.
    if (message) {
      throw new Error(message)
    }
```

```
  // 3. Пакуем результаты в один объект.
    let latestUserInfo = {
      name,
      githubLogin: login,
      githubToken: access_token,
      avatar: avatar_url
    }
  // 4. Добавляем новую информацию или обновляем запись.
    const { ops:[user] } = await db
      .collection('users')
      .replaceOne({ githubLogin: login }, latestUserInfo,
      { upsert: true })
  // 5. Возвращаем данные пользователя и его токен.
    return { user, token: access_token }

}
```

Распознаватели могут быть асинхронными. Мы можем дождаться ответа по сети, прежде чем вернуть результат операции клиенту. Распознаватель `githubAuth` является асинхронным, потому что мы должны дождаться двух ответов от GitHub, прежде чем у нас появятся данные, которые нам нужно вернуть.

После того как мы получили данные пользователя от GitHub, мы проверяем нашу локальную базу данных, чтобы увидеть, входил ли этот пользователь в прошлом в наше приложение, что означает, что у него уже есть учетная запись. Если у пользователя есть учетная запись, мы обновим ее данные информацией, полученной нами от GitHub. Возможно, он изменил свое имя или изображение профиля со времени последнего входа в систему. Если у пользователя еще нет учетной записи, мы добавим нового пользователя в нашу коллекцию пользователей. В обоих случаях мы возвращаем зарегистрированные `user` и `token` из этого распознавателя.

Пришло время протестировать данный процесс авторизации, и для тестирования вам нужен код. Чтобы получить код, вам необходимо добавить свой идентификатор клиента по этому URL-адресу:

```
https://github.com/login/oauth/authorize?client_id=ВАШ-ID&scope=user
```

Вставьте URL-адрес с указанием вашего `client_id` из GitHub в адресной строке нового окна браузера. Вы будете перенаправлены на сайт GitHub, где нужно согласиться авторизовать данное приложение. Когда вы авторизуете приложение, GitHub перенаправит вас обратно на адрес http://localhost:3000 с кодом:

```
http://locahost:3000?code=XYZ
```

В нашем примере код — это `XYZ`. Скопируйте код из URL-адреса браузера, а затем отправьте его с помощью мутации `githubAuth`:

```
mutation {
  githubAuth(code:"XYZ") {
    token
    user {
      githubLogin
      name
      avatar
    }
  }
}
```

Данная мутация позволит авторизовать текущего пользователя и вернуть токен вместе с информацией об этом пользователе. Сохраните токен. Нам нужно будет отправить его в заголовке в будущих запросах.

Неверные учетные данные

Когда вы видите ошибку «Неверные учетные данные» (Bad Credentials), это означает, что идентификатор клиента, пароль клиента или код, который был отправлен в API GitHub, неверен. Проверьте идентификатор клиента и пароль клиента. Часто это код, который вызывает данную ошибку.

Коды GitHub действуют только в течение ограниченного периода времени и могут использоваться лишь один раз. Если в распознавателе произошла ошибка после запроса учетных данных, код, применяемый в запросе, больше не может быть использован. Как правило, эту ошибку можно устранить, запросив другой код из GitHub.

Аутентификация пользователей

Чтобы идентифицировать себя в будущих запросах, вам нужно отправить токен с каждым запросом в заголовке `Authorization`. Этот токен будет применяться для идентификации пользователя путем поиска его записи в базе данных.

На платформе GraphQL Playground вы можете добавлять заголовки к каждому запросу. Перейдите на вкладку HTTP Headers (Заголовки HTTP) (рядом с вкладкой Query Variables (Переменные запроса)). Вы можете добавить HTTP-заголовки к запросу на этой вкладке. Просто отправьте заголовки в формате JSON:

```
{
  "Authorization": "<ВАШ_ТОКЕН>"
}
```

Замените заполнитель `<ВАШ_ТОКЕН>` токеном, который был возвращен из мутации `githubAuth`. Теперь вы отправляете ключ к вашей идентификации с каждым запросом GraphQL. Нам нужно использовать этот ключ, чтобы найти свою учетную запись и добавить ее в контекст.

Запрос me

Теперь нам нужно создать запрос, относящийся к нашей собственной пользовательской информации: `me`. Этот запрос возвращает текущего зарегистрированного пользователя на основе токена, отправленного в HTTP-заголовках запроса. Если в настоящий момент пользователь не зарегистрирован, запрос возвращает `null`.

Процесс начинается, когда клиент отправляет GraphQL-запрос, `me`, с `Authorization: token` для безопасности информации пользователя. Затем API захватывает заголовок `Authorization` и применяет токен для поиска текущей записи пользователя в базе данных. Он также добавляет текущую учетную запись пользователя в контекст. После этого каждый распознаватель будет иметь доступ к текущему пользователю.

Мы должны идентифицировать текущего пользователя и поместить его в контекст. Изменим конфигурацию нашего сервера.

Придется изменить способ создания объекта контекста. Вместо объекта мы будем применять функцию для обработки контекста:

```
async function start() {
    const app = express()
    const MONGO_DB = process.env.DB_HOST

    const client = await MongoClient.connect(
      MONGO_DB,
      { useNewUrlParser: true }
    )

    const db = client.db()

    const server = new ApolloServer({
      typeDefs,
      resolvers,
      context: async ({ req }) => {
        const githubToken = req.headers.authorization
        const currentUser =
        await db.collection('users').findOne({ githubToken })
        return { db, currentUser }
      }
    })

    ...

}
```

Контекст может быть объектом или функцией. Чтобы наше приложение работало, нам нужно, чтобы он был функцией, чтобы мы могли устанавливать контекст каждый раз, когда есть запрос. Когда контекст является функцией, он включается в каждый GraphQL-запрос. Объектом, который возвращается этой функцией, будет контекст, который отправляется в распознаватель.

В функции контекста мы можем захватить заголовок авторизации из запроса и проанализировать его на предмет токена. После того как у нас есть токен, мы можем применить его для поиска пользователя в нашей базе данных. Если у нас есть пользователь, он будет добавлен в контекст. Если нет, значение для пользователя в контексте будет равно `null`.

С помощью этого кода настало время добавить запрос `me`. В первую очередь нам нужно изменить нашу схему `typeDefs`:

```
type Query {
  me: User
  ...
}
```

Запрос `me` возвращает пользователя, который может иметь значение `null`. Он будет `null`, если текущий авторизованный пользователь не найден. Добавим распознаватель для запроса `me`:

```
const resolvers = {
  Query: {
    me: (parent, args, { currentUser }) => currentUser,
    ...
  }
}
```

Мы уже проделали тяжелый путь поиска пользователя на основе его токена. На текущем этапе вы просто вернете объект `currentUser` из контекста. Опять же это будет `null`, если пользователя нет.

Если правильный токен был добавлен в заголовок авторизации HTTP, вы можете отправить запрос, чтобы получить информацию о себе, используя запрос `me`:

```
query currentUser {
  me {
    githubLogin
    name
    avatar
  }
}
```

Когда вы запустите данный запрос, вы будете идентифицированы. Хороший тест, подтверждающий, что все правильно, — это попытаться запустить указанный запрос без заголовка авторизации или с неправильным токеном. Учитывая неправильный токен или отсутствующий заголовок, вы должны увидеть, что запрос `me` равен `null`.

Мутация postPhoto

Чтобы отправить фотографию в наше приложение, пользователь должен авторизоваться. Мутация `postPhoto` может определить, кто авторизуется, проверяя контекст. Изменим мутацию `postPhoto`:

```
async postPhoto(parent, args, { db, currentUser }) {

  // 1. Если в контексте нет пользователя, выбросить ошибку.
  if (!currentUser) {
      throw new Error('only an authorized user can post a photo')
  }

  // 2. Сохранить идентификатор текущего пользователя
  // с фотографией.
  const newPhoto = {
      ...args.input,
      userID: currentUser.githubLogin,
      created: new Date()
  }

  // 3. Вставить новую фотографию, записать идентификатор,
  // созданный базой данных.
  const { insertedIds } =
  await db.collection('photos').insert(newPhoto)
  newPhoto.id = insertedIds[0]

  return newPhoto

}
```

Мутация `postPhoto` претерпела несколько изменений, чтобы можно было сохранить новую фотографию в базе данных. В первую очередь значение `currentUser` извлекается из контекста. Если оно равно `null`, выбрасывается ошибка и предотвращается выполнение последующей мутации `postPhoto`. Чтобы отправить фотографию, пользователь должен отправить правильный токен в заголовке `Authorization`.

Затем добавим идентификатор текущего пользователя к объекту `newPhoto`. Теперь мы можем сохранить новую запись фото

в коллекции фотографий в базе данных. Mongo создает уникальный идентификатор для каждого сохраненного документа. Когда новая фотография добавляется, мы можем получить этот идентификатор, задействуя массив `insertIds`. Прежде чем мы вернем фотографию, нам нужно убедиться, что она имеет уникальный идентификатор.

Нам также нужно изменить распознаватели `Photo`:

```
const resolvers = {
  ...
  Photo: {
    id: parent => parent.id || parent._id,
    url: parent => `/img/photos/${parent._id}.jpg`,
    postedBy: (parent, args, { db }) =>
      db.collection('users').findOne({ githubLogin: parent.userID
  })
}
```

Если клиент запрашивает идентификатор фотографии, мы должны убедиться, что он получает правильное значение. Если родительская фотография еще не имеет идентификатора, мы можем предположить, что для нее была создана запись базы данных и она будет иметь идентификатор, сохраненный в поле `_id`. Мы должны убедиться, что поле идентификатора фотографии связано с идентификатором базы данных.

Затем предположим, что мы обслуживаем эти фотографии с одного и того же веб-сервера. Мы возвращаем локальный маршрут к фотографии. Данный локальный маршрут создается с применением идентификатора фотографии.

Наконец, нам нужно изменить распознаватель `postedBy`, чтобы найти пользователя, разместившего фотографию в базе данных. Мы можем задействовать `userID`, который сохраняется вместе с родительской фотографией, для поиска записи этого пользователя в базе данных. Идентификатор пользователя фотографии должен соответствовать значению `githubLogin` пользователя, поэтому метод `.findOne()` должен возвращать одну пользовательскую запись — пользователя, разместившего фотографию.

С нашим заголовком авторизации мы должны иметь возможность отправлять новые фотографии в сервис GraphQL:

```
mutation post($input: PostPhotoInput!) {
  postPhoto(input: $input) {
    id
    url
    postedBy {
      name
      avatar
    }
  }
}
```

После того как мы разместим фотографию, мы можем запросить ее идентификатор и URL-адрес вместе с `name` и `avatar` пользователя, добавившего фотографию.

Мутация поддельных пользователей

Чтобы протестировать наше приложение с различными пользователями, добавим мутацию, которая позволит нам заполнить базу данных поддельными пользователями из API `random.me`.

Мы можем сделать это с помощью мутации `addFakeUsers`. Сначала добавим ее в схему:

```
type Mutation {
  addFakeUsers(count: Int = 1): [User!]!
  ...
}
```

Обратите внимание, что аргумент счетчика принимает количество поддельных пользователей для добавления и возвращает список пользователей. Этот список содержит учетные записи поддельных пользователей, добавленных данной мутацией. По умолчанию мы добавляем одного пользователя за раз, но вы можете добавить больше, отправив этой мутации другой счетчик:

```
addFakeUsers: async (root, {count}, {db}) => {

    var randomUserApi = `https://randomuser.me/api/
    ?results=${count}`

    var { results } = await fetch(randomUserApi)
```

```
      .then(res => res.json())

    var users = results.map(r => ({
      githubLogin: r.login.username,
      name: `${r.name.first} ${r.name.last}`,
      avatar: r.picture.thumbnail,
      githubToken: r.login.sha1
    }))

    await db.collection('users').insert(users)

    return users

}
```

Чтобы проверить добавление новых пользователей, сначала мы должны получить некоторые поддельные данные из `randomuser.me.addFakeUsers` — асинхронной функции, которую мы можем применять для извлечения указанных данных. Потом мы сериализуем данные из `randomuser.me`, создавая объекты пользователя, соответствующие нашей схеме. Затем мы добавляем этих новых пользователей в базу данных и возвращаем список новых пользователей.

Теперь мы можем заполнить базу данных с помощью мутации:

```
mutation {
  addFakeUsers(count: 3) {
    name
  }
}
```

Эта мутация добавляет трех поддельных пользователей в базу данных. Теперь, когда у нас есть поддельные пользователи, мы можем авторизоваться с помощью поддельной учетной записи пользователя через мутацию. Добавим `fakeUserAuth` к нашему типу `Mutation`:

```
type Mutation {
  fakeUserAuth(githubLogin: ID!): AuthPayload!
  ...
}
```

Затем нужно добавить распознаватель, возвращающий токен, который мы можем применить для авторизации наших поддельных пользователей:

```
async fakeUserAuth (parent, { githubLogin }, { db }) {

  var user = await db.collection('users').findOne({ githubLogin })

  if (!user) {
      throw new Error(`Cannot find user with githubLogin
      "${githubLogin}"`)
  }

  return {
      token: user.githubToken,
      user
  }
}
```

Распознаватель `fakeUserAuth` получает `githubLogin` из аргументов мутации и применяет его для поиска этого пользователя в базе данных. После того как он найдет этого пользователя, токен и учетная запись последнего будут возвращены в форме нашего типа `AuthPayload`.

Теперь мы можем аутентифицировать поддельных пользователей, отправив мутацию:

```
mutation {
  fakeUserAuth(githubLogin:"jDoe") {
    token
  }
}
```

Добавьте возвращенный токен в HTTP-заголовок авторизации, чтобы публиковать новые фотографии от имени этого поддельного пользователя.

Резюме

Итак, вы создали сервер GraphQL. Начав с тщательного рассмотрения распознавателей, вы перешли к обработке запросов и мутациям. Вы добавили авторизацию с помощью аккаунта на сайте GitHub. Идентифицировали текущего пользователя через токен доступа, который добавляется в заголовок каждого запроса. И наконец, вы изменили мутацию, читающую пользователя из контекста распознавателя и позволяющую публиковать фотографии.

Если вы хотите запустить окончательную версию сервиса, которую создали в данной главе, найдите ее в репозитории книги (https://github.com/MoonHighway/learning-graphql/tree/master/chapter-05/photo-share-api/). Это приложение должно знать, какую базу данных и какие учетные данные GitHub OAuth использовать. Вы можете добавить эти значения, создав новый файл с расширением `.env` и поместив его в корень проекта:

```
DB_HOST=<ХОСТ_MONGODB>
CLIENT_ID=<ID_КЛИЕНТА_GITHUB>
CLIENT_SECRET=<ПАРОЛЬ_КЛИЕНТА_GITHUB>
```

С помощью файла `.env` вы можете установить зависимости: `yarn` или `npm install` — и запустить сервис: `yarn start` или `npm start`. Как только сервис запущен на порте 4000, вы можете отправлять ему запросы с помощью платформы Playground по адресу http://localhost:4000/playground. Вы можете запросить GitHub-код, щелкнув на ссылке http://localhost:4000. Если хотите получить доступ к конечной точке GraphQL у другого клиента, перейдите по адресу http://localhost:4000/graphql.

В главе 7 мы расскажем, как изменить API для обработки подписок и загрузок файлов. Но прежде, чем мы это сделаем, нужно показать, как клиенты будут использовать данный API, поэтому в главе 6 мы поработаем со стороной клиента для поддержки услуги.

6 Клиенты GraphQL

Теперь, когда сервер GraphQL сконфигурирован, настало время настроить GraphQL на стороне клиента. В целом клиент — это просто приложение, которое общается с сервером. Из-за гибкости GraphQL нет конкретного рецепта того, как создать клиент. Возможно, вы создаете приложения для браузеров. Или для смартфонов. Или для дисплея на вашем холодильнике. Не имеет значения для клиента и то, на каком языке он написан.

Все, что вам нужно, чтобы отправлять запросы и мутации, — это реализовать возможность отправлять HTTP-запросы. Когда сервис отвечает некоторыми данными, вы можете использовать их в своем клиенте независимо от того, что это за клиент.

API GraphQL

Самый простой способ — отправить HTTP-запрос конечной точке GraphQL. Чтобы протестировать сервер, который мы создали в главе 5, убедитесь, что ваш сервис запущен локально по адресу http://localhost:4000/graphql. Вы также можете найти все эти примеры, запущенные на сайте CodeSandbox, по ссылкам, указанным в репозитории главы 6 (github.com/MoonHighway/learning-graphql/tree/master/chapter-06).

Запросы на выборку

Как вы видели в главе 3, можно отправлять запросы в сервис GraphQL с помощью cURL. Нужно лишь несколько разных значений:

- запрос — `{totalPhotos, totalUsers}`;
- конечная точка GraphQL — `http://localhost:4000/graphql`;
- тип содержимого — `Content-Type: application/json`.

Теперь мы отправляем запрос cURL непосредственно из оболочки командной строки, используя метод POST:

```
curl -X POST \
     -H "Content-Type: application/json" \
     --data '{ "query": "{totalUsers, totalPhotos}" }' \
     http://localhost:4000/graphql
```

Если мы отправим данный запрос, то должны увидеть корректные результаты, `{"data":{"totalUsers":7, "totalPhotos":4}}`, поскольку данные JSON возвращаются в оболочку командной строки. Переменным `totalUsers` и `totalPhotos` будут присвоены наши текущие данные. Если наш клиент является сценарием оболочки, можно создать этот сценарий с помощью cURL.

Поскольку мы используем cURL, то можем задействовать все, что отправляет HTTP-запрос. Мы могли бы с помощью объекта `fetch` построить маленький клиент, который будет работать в браузере:

```
var query = `{totalPhotos, totalUsers}`
var url = 'http://localhost:4000/graphql'

var opts = {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({ query })
}

fetch(url, opts)
  .then(res => res.json())
  .then(console.log)
  .catch(console.error)
```

После получения данных мы увидим ожидаемый результат в оболочке командной строки:

```
{
  "data": {
    "totalPhotos": 4,
    "totalUsers": 7
  }
}
```

Мы можем использовать полученные данные для клиента при создании приложений. Рассмотрим основной пример того, как можно перечислять `totalUsers` и `totalPhotos` непосредственно в DOM:

```
fetch(url, opts)
  .then(res => res.json())
  .then(({data}) => `
        <p>photos: ${data.totalPhotos}</p>
        <p>users: ${data.totalUsers}</p>
  `)
  .then(text => document.body.innerHTML = text)
  .catch(console.error)
```

Вместо вывода результатов в оболочке командной строки мы используем данные для генерации некоторого HTML-кода. Затем мы можем взять этот код и записать его прямо в тело документа. Обратите внимание: после завершения запроса можно перезаписать все, что было в теле.

Если вы знаете, как отправлять HTTP-запросы, применяя свой клиент, у вас уже есть инструменты, необходимые для создания клиентского приложения, взаимодействующего с любым API GraphQL.

Инструмент graphql-request

Хотя cURL и fetch великолепны, существуют другие фреймворки, которые можно использовать для передачи операций GraphQL в API. Один из наиболее примечательных — `graphql-request`. Он оберты-вает запросы на выборку в промис, который может быть применен для запросов на сервер GraphQL. Кроме того, он обрабатывает детали запроса и выполняет анализ данных.

Чтобы начать работу с `graphql-request`, сначала нужно установить его:

```
npm install graphql-request
```

Теперь мы импортируем и используем модуль в качестве запроса. Нужно убедиться, что сервис фотографий работает на порте 4000:

```
import { request } from 'graphql-request'

var query = `
  query listUsers {
    allUsers {
      name
      avatar
    }
  }
`

request('http://localhost:4000/graphql', query)
    .then(console.log)
    .catch(console.error)
```

Функция запроса принимает `url` и `query`, выполняет запрос на сервер и возвращает данные в одной строке кода. Возвращенные данные, как и ожидалось, соответствуют JSON-ответам всех пользователей:

```
{
  "allUsers": [
    { "name": "sharon adams", "avatar": "http://..." },
    { "name": "sarah ronau", "avatar": "http://..." },
    { "name": "paul young", "avatar": "http://..." },
  ]
}
```

Можно сразу применять эти данные в нашем клиенте.

Также можно отправлять мутации с помощью `graphql-request`:

```
import { request } from 'graphql-request'

var url = 'http://localhost:4000/graphql'

var mutation = `
```

```
    mutation populate($count: Int!) {
        addFakeUsers(count:$count) {
            id
            name
        }
    }
`

var variables = { count: 3 }

request(url, mutation, variables)
    .then(console.log)
    .catch(console.error)
```

Функция `request` принимает URL-адрес API, мутацию и третий аргумент для переменных. Это всего лишь JavaScript-объект, который передает поле и значение для переменных запроса. После вызова `request` мы выдаем мутацию `addFakeUsers`.

Хотя `graphql-request` не подразумевает никакой официальной интеграции с библиотеками и фреймворками пользовательского интерфейса, мы можем легко интегрировать библиотеку. Загрузим некоторые данные в компонент React, задействуя `graphql-request`, как показано в примере 6.1.

Пример 6.1. GraphQL-запрос и React

```
import React from 'react'
import ReactDOM from 'react-dom'
import { request } from 'graphql-request'

var url = 'http://localhost:4000/graphql'

var query = `
  query listUsers {
    allUsers {
      avatar
      name
    }
  }
`

var mutation = `
```

```
    mutation populate($count: Int!) {
        addFakeUsers(count:$count) {
            githubLogin
        }
    }
`

const App = ({ users=[] }) =>
    <div>
        {users.map(user =>
            <div key={user.githubLogin}>
                <img src={user.avatar} alt="" />
                {user.name}
            </div>
        )}
        <button onClick={addUser}>Add User</button>
    </div>

const render = ({ allUsers=[] }) =>
    ReactDOM.render(
        <App users={allUsers} />,
        document.getElementById('root')
    )

const addUser = () =>
    request(url, mutation, {count:1})
        .then(requestAndRender)
        .catch(console.error)

const requestAndRender = () =>
    request(url, query)
        .then(render)
        .catch(console.error)

requestAndRender()
```

Код в этом файле начинается с команд импорта как React, так и ReactDOM. Затем мы создаем компонент `App`. `App` отображает `users`, которые передаются как свойства, и создает элементы `div`, содержащие их `avatar` и `username`. Функция `render` отображает `App` в элемент `#root` и передается в `allUsers` как свойство.

Теперь `requestAndRender` вызывает `request` из `graphql-request`. Так нам предоставляется запрос, получение данных, а затем вызов `render`, который предоставляет данные компоненту `App`.

Это небольшое приложение также поддерживает мутации. В компоненте `App` у кнопки есть событие `onClick`, которое вызывает функцию `addUser`. При вызове отправляется мутация, а затем вызывается `requestAndRender` для выдачи нового запроса для обслуживания пользователей и отображения `<App/>` с новым списком пользователей.

До сих пор мы рассматривали несколько способов начать создание клиентских приложений с помощью GraphQL. Вы можете писать сценарии оболочки с помощью cURL. Создавать веб-страницы с помощью фреймворка fetch. Создавать приложения быстрее с помощью `graphql-request`. Вы могли бы остановиться прямо здесь, если очень хочется, но есть еще более мощные клиенты GraphQL. Взглянем на них.

Apollo Client

Огромным преимуществом использования передачи состояния представления (Representational State Transfer, REST) является простота, с которой вы можете справиться с кэшированием. С помощью REST вы можете сохранить данные ответа из запроса в кэше по URL-адресу, который применялся для доступа к этому запросу. Кэширование выполнено, никаких проблем. Кэширование GraphQL немного сложнее. У нас нет множества маршрутов с API GraphQL — все отправляется и принимается одной конечной точкой, поэтому мы не можем просто сохранять данные, возвращенные из маршрута по URL-адресу, который использовался для его запроса.

Чтобы создать надежное, эффективное приложение, нам нужен способ кэширования запросов и объектов их результатов. Наличие локализованного кэширующего решения очень важно, поскольку мы постоянно стремимся создавать быстрые и эффективные приложения. Мы могли бы создать что-то подобное сами или опереться на один из проверенных клиентов, которые уже существуют.

Наиболее известными клиентскими решениями GraphQL, доступными сегодня, являются Relay и Apollo Client. Relay был представлен с открытым исходным кодом компанией Facebook в 2015 году одновременно с GraphQL. Он объединяет весь опыт компании Facebook, касающийся применения GraphQL в рабочей среде. Relay совместим только с React и React Native, что означает: появилась возможность создать клиент GraphQL для тех разработчиков, которые не могут использовать React.

Знакомьтесь с Apollo Client. Apollo Client — это проект гибкого клиентского решения GraphQL для выполнения таких процедур, как кэширование, обновления пользовательского интерфейса и многое другое. Команда разработчиков создала пакеты, которые предоставляют связки для React, Angular, Ember, Vue, iOS и Android.

Мы уже задействовали несколько инструментов команды разработчиков Apollo на сервере, но Apollo Client фокусируется на отправке запросов от клиента на сервер и их получении. Он обрабатывает сетевые запросы с помощью Apollo Link и выполняет все кэширование с помощью Apollo Cache. Затем Apollo Client завершает связь и эффективно управляет всеми взаимодействиями с сервисом GraphQL.

В оставшейся части главы мы подробно рассмотрим Apollo Client. Мы будем задействовать React для создания компонентов нашего пользовательского интерфейса, но можем применить многие из описанных здесь методов к проектам, использующим разные библиотеки и фреймворки.

Apollo Client и React

Поскольку работа с React привела нас к GraphQL, мы выбрали React как библиотеку пользовательского интерфейса. Мы не рассказали о самой технологии React. Это библиотека, созданная в Facebook, которая применяет компонентную архитектуру для составления пользовательских интерфейсов. Если вы предпочитаете другую библиотеку и не хотите вновь касаться React, мы вас понимаем. Идеи, представленные в следующем разделе, применимы к другим структурам пользовательских фреймворков.

Настройка проекта

В этой главе мы рассмотрим, как создать приложение React, которое взаимодействует с сервисом GraphQL с помощью Apollo Client. Для начала нам нужно обеспечить интерфейс данного проекта с помощью инструмента `create-react-app`. Он позволяет создавать весь проект React без настройки какой-либо конфигурации сборки. Если раньше вы не использовали `create-react-app`, нужно будет установить его:

```
npm install -g create-react-app
```

После установки вы можете создать проект React в любом расположении на вашем компьютере с помощью следующей команды:

```
create-react-app photo-share-client
```

Эта команда устанавливает новое базовое приложение React в папку `photo-share-client`. Она автоматически добавляет и устанавливает все необходимое, чтобы приступить к созданию приложения React. Чтобы запустить приложение, перейдите в папку `photo-share-client` и выполните команду `npm start`. В браузере откроется страница http://localhost:3000, где будет запущено ваше клиентское приложение React. Помните: вы можете найти все файлы этой главы в репозитории по адресу github.com/moonhighway/learning-graphql.

Конфигурирование Apollo Client

Вам необходимо установить несколько пакетов для создания клиента GraphQL с помощью инструментов Apollo. В первую очередь нужен `graphql`, который включает в себя парсер языка GraphQL. Понадобится пакет под названием `apollo-boost`. Apollo Boost включает пакеты Apollo, необходимые для создания клиента Apollo и отправки операций клиенту. Наконец, понадобится `react-apollo`. React Apollo — это библиотека npm, содержащая компоненты React, которые мы будем задействовать для создания пользовательского интерфейса с помощью Apollo.

Установим три данных пакета одновременно:

```
npm install graphql apollo-boost react-apollo
```

Теперь мы готовы создать клиент. Конструктор `ApolloClient`, расположенный в `apollo-boost`, может быть использован для создания нашего первого клиента. Откройте файл `src/index.js` и замените код в нем следующим:

```
import ApolloClient from 'apollo-boost'

const client = new ApolloClient({
  uri: 'http://localhost:4000/graphql' })
```

Используя конструктор `ApolloClient`, мы создали новый экземпляр `client`. `client` готов обрабатывать все сетевые взаимодействия с помощью сервиса GraphQL, размещенного по адресу http://localhost:4000/graphql. Например, мы можем задействовать клиент для отправки запроса в сервис PhotoShare:

```
import ApolloClient, { gql } from 'apollo-boost'

const client = new ApolloClient({ uri:
  'http://localhost:4000/graphql' })

const query = gql`
    {
        totalUsers
        totalPhotos
    }
`


client.query({query})
    .then(({ data }) => console.log('data', data))
    .catch(console.error)
```

Данный код применяет `client` для отправки запроса общего количества фотографий и общего количества пользователей. Чтобы это произошло, мы импортировали функцию `gql` из `apollo-boost`. Указанная функция является частью пакета `graphql-tag`, который автоматически включен в `apollo-boost`. Функция `gql` используется для анализа запроса в дереве синтаксиса АСД или дерева абстрактного синтаксиса.

Мы можем отправить АСД к клиенту, вызвав метод `client.query({query})`. Этот метод возвращает промис. Он отправляет запрос

в виде HTTP в наш сервис GraphQL и разбирает данные, возвращенные этим сервисом. В приведенном выше примере мы выводим ответ в оболочку командной строки:

```
{ totalUsers: 4, totalPhotos: 7, Symbol(id): "ROOT_QUERY" }
```

Сервис GraphQL должен работать

Убедитесь, что сервис GraphQL по-прежнему работает по адресу http://localhost:4000, чтобы вы могли протестировать соединение клиента с сервером.

В дополнение к обработке всех сетевых запросов к нашему сервису GraphQL клиент также кэширует ответы локально в памяти. В любой момент мы можем взглянуть на кэш, вызвав функцию `client.extract()`:

```
console.log('cache', client.extract())
client.query({query})
    .then(() => console.log('cache', client.extract()))
    .catch(console.error)
```

Здесь мы рассмотрим кэш перед отправкой запроса, а затем взглянем на него после того, как запрос был разобран. Мы видим, что теперь у нас есть результаты, сохраненные в локальном объекте, который управляется клиентом:

```
{
    ROOT_QUERY: {
        totalPhotos: 4,
        totalUsers: 7
    }
}
```

В следующий раз, когда мы отправим клиенту запрос на эти данные, он будет считывать его из кэша, а не отправлять другой сетевой запрос в наш сервис. Apollo Client предоставляет нам возможность указывать, когда и как часто мы должны отправлять HTTP-запросы по сети. Мы рассмотрим указанные варианты позже в настоящей главе. На данный момент важно понимать, что Apollo Client используется для обработки всех сетевых запросов в нашем сервисе GraphQL.

Кроме того, по умолчанию он автоматически кэширует результаты локально и отдает предпочтение локальному кэшу, чтобы улучшить производительность наших приложений.

Чтобы начать работу с `react-apollo`, нам нужно лишь создать клиент и добавить его в наш пользовательский интерфейс с компонентом `ApolloProvider`. Замените код в файле `index.js` следующим:

```
import React from 'react'
import { render } from 'react-dom'
import App from './App'
import { ApolloProvider } from 'react-apollo'
import ApolloClient from 'apollo-boost'

const client = new ApolloClient({ uri:
  'http://localhost:4000/graphql' })

render(
    <ApolloProvider client={client}>
      <App />
    </ApolloProvider>,
    document.getElementById('root')
)
```

Это весь код, который необходим для взаимодействия Apollo с React. Мы создали клиент, а затем разместили его в глобальной области React с помощью компонента, называемого `ApolloProvider`. Любой дочерний компонент, обернутый `ApolloProvider`, будет иметь доступ к клиенту. Это означает, что компонент `<App />` и любой из его дочерних элементов готов принимать данные из нашего сервиса GraphQL через Apollo Client.

Компонент Query

При применении Apollo Client нам нужен способ обработки запросов для сбора данных для загрузки в наш интерфейс React. Компонент `Query` позаботится об извлечении данных, обработке состояния загрузки и обновлении нашего пользовательского интерфейса. Мы можем применить компонент `Query` в любой позиции `ApolloProvider`. Компонент `Query` отправляет `query` с использованием клиента. После

разрешения клиент вернет результаты, которые мы будем применять для создания пользовательского интерфейса.

Откройте файл `src/App.js` и поменяйте код в нем следующим образом:

```
import React from 'react'
import Users from './Users'
import { gql } from 'apollo-boost'

export const ROOT_QUERY = gql`
    query allUsers {
        totalUsers
        allUsers {
            githubLogin
            name
            avatar
        }
    }
`

const App = () => <Users />

export default App
```

В компоненте `App` мы создали запрос `ROOT_QUERY`. Помните, что одним из преимуществ применения GraphQL является запрос всего, что нужно для создания вашего пользовательского интерфейса и получения всех указанных данных в одном ответе. Это означает, что мы будем запрашивать как `totalUsers`, так и массив `allUsers` в запросе, который мы создали в корне нашего приложения. Используя функцию `gql`, мы преобразовали наш строковый запрос в объект АСД с именем `ROOT_QUERY` и экспортировали этот объект, чтобы его могли применять другие компоненты.

На данном этапе вы должны увидеть ошибку. Это потому, что мы инструктировали `App` отобразить компонент, который еще не создали. Создайте файл `src/Users.js` и поместите следующий код в него:

```
import React from 'react'
import { Query } from 'react-apollo'
import { ROOT_QUERY } from './App'

const Users = () =>
```

```
    <Query query={ROOT_QUERY}>
        {result =>
           <p>Users are loading: {result.loading ? "yes" : "no"}</p>
        }
    </Query>

export default Users
```

Теперь вы должны увидеть явное сообщение об ошибке, и в окне браузера должно отображаться сообщение Users are loading: no. За кадром компонент `Query` отправляет `ROOT_QUERY` в наш сервис GraphQL и кэширует результат локально. Мы получаем результат с использованием такой методики React, как свойства визуализации. Свойства визуализации позволяют передавать свойства в качестве аргументов функции дочерним компонентам. Обратите внимание, что мы получаем `result` из функции и возвращаем элемент абзаца.

Результат содержит больше информации, чем лишь данные в ответе. Он расскажет нам, загружается ли операция через свойство `result.loading`. В предыдущем примере мы можем сообщить пользователю, загружается текущий запрос или нет.

Отмена HTTP-запроса

Ваша сеть может быть слишком быстрой, и вы не увидите процесс загрузки в браузере. Вы можете применять вкладку Network (Сеть) в инструментах разработчика браузера Chrome для отслеживания HTTP-запроса. В инструментах разработчика вы найдете раскрывающийся список, в котором есть команда Online (Интернет). Выберите пункт Slow 3G (Медленный 3G), чтобы замедлить обмен данными и увидеть процесс загрузки в браузере.

Как только данные загрузятся, они будут переданы вместе с результатом.

Вместо того чтобы показывать «да» или «нет», когда клиент загружает данные, мы можем отображать компоненты пользовательского интерфейса. Настроим файл `Users.js`:

```
const Users = () =>
    <Query query={ROOT_QUERY}>
```

```
        {({ data, loading }) => loading ?
            <p>loading users...</p> :
            <UserList count={data.totalUsers}
            users={data.allUsers} />
        }
    </Query>

const UserList = ({ count, users }) =>
    <div>
        <p>{count} Users</p>
        <ul>
            {users.map(user =>
                <UserListItem key={user.githubLogin}
                    name={user.name}
                    avatar={user.avatar} />
            )}
        </ul>
    </div>

const UserListItem = ({ name, avatar }) =>
    <li>
        <img src={avatar} width={48} height={48} alt="" />
        {name}
    </li>
```

Если клиент *загружает* текущий запрос, будет выведено сообщение loading users.... Если данные были загружены, мы передадим общее количество пользователей вместе с массивом, содержащим `name`, `githubLogin` и `avatar` каждого пользователя для компонента `UserList`: именно те данные, которые мы указывали в нашем запросе. `UserList` задействует данные результата для создания пользовательского интерфейса. Он выводит счетчик вместе со списком, в котором отображаются аватары и имена пользователей.

Объект `results` также имеет несколько полезных функций для пагинации, повторного выбора и опроса. Применим функцию `refetch`, чтобы повторно запросить список пользователей, когда мы нажимаем кнопку:

```
const Users = () =>
    <Query query={ROOT_QUERY}>
        {({ data, loading, refetch }) => loading ?
```

```
            <p>loading users...</p> :
            <UserList count={data.totalUsers}
                users={data.allUsers}
                refetchUsers={refetch} />
        }
    </Query>
```

Здесь мы получили функцию, которая может быть применена для повторного обновления `ROOT_QUERY` или запроса данных с сервера. Свойство `refetch` — это просто функция. Мы можем передать его в `UserList`, где его можно добавить к событию нажатия кнопки:

```
const UserList = ({ count, users, refetch }) =>
    <div>
        <p>{count} Users</p>
        <button onClick={() => refetch()}>Refetch</button>
        <ul>
            {users.map(user =>
                <UserListItem key={user.githubLogin}
                    name={user.name}
                    avatar={user.avatar} />
            )}
        </ul>
    </div>
```

В `UserList` мы применяем функцию `refetch` для запроса тех же корневых данных из нашего сервиса GraphQL. Всякий раз, когда вы нажимаете кнопку Refetch Users, другой запрос будет отправлен в конечную точку GraphQL для обновления любых изменений данных. Это один из способов поддерживать синхронизацию пользовательского интерфейса с данными на сервере.

Чтобы проверить это, мы можем изменить данные пользователя после первоначальной выборки. Вы можете удалить коллекцию пользователей, удалить пользовательские документы непосредственно из MongoDB или добавить поддельных пользователей, отправив запрос на сервер GraphQL Playground. При изменении данных в базе данных необходимо щелкнуть на кнопке Refetch Users, чтобы получить самые последние данные в браузере.

Опрос — еще один вариант, доступный с помощью компонента `Query`. Когда мы добавляем свойство `pollInterval` к компоненту `Query`, данные автоматически извлекаются снова и снова на основе заданного интервала:

```
<Query query={ROOT_QUERY} pollInterval={1000}>
```

Установка `pollInterval` автоматически включает переполучение данных через указанное время. В этом случае мы будем обновлять данные с сервера каждую секунду. Будьте внимательны при использовании опроса, поскольку данный код на самом деле посылает новый сетевой запрос каждую секунду.

В дополнение к `loading`, `data` и `refetch` объект ответа имеет несколько дополнительных параметров.

- `stopPolling` — функция, останавливающая опрос.
- `startPolling` — функция, запускающая опрос.
- `fetchMore` — функция, которая может быть использована для получения следующей страницы данных.

Прежде чем продолжить, удалите свойства `pollInterval` из компонента `Query`. Нам не нужно, чтобы опрос происходил по мере продолжения итераций этого примера.

Компонент Mutation

Когда нужно отправить мутации в сервис GraphQL, мы можем применить компонент `Mutation`. В следующем примере мы задействуем данный компонент для обработки мутации `addFakeUsers`. Отправляя эту мутацию, мы записываем новый список пользователей непосредственно в кэш.

Для начала импортируем компонент `Mutation` и добавим мутацию в файл `Users.js`:

```
import { Query, Mutation } from 'react-apollo'
import { gql } from 'apollo-boost'

...

const ADD_FAKE_USERS_MUTATION = gql`
```

```
    mutation addFakeUsers($count:Int!) {
        addFakeUsers(count:$count) {
            githubLogin
            name
            avatar
        }
    }
`
```

Получив мутацию, мы можем использовать ее в комбинации с компонентом `Mutation`. Этот компонент передаст функцию своим дочерним элементам через свойства визуализации. Данная функция может применяться для отправки мутации при готовности:

```
const UserList = ({ count, users, refetchUsers }) =>
    <div>
        <p>{count} Users</p>
        <button onClick={() => refetchUsers()}>Refetch
        Users</button>
        <Mutation mutation={ADD_FAKE_USERS_MUTATION}
        variables={{ count: 1 }}>
            {addFakeUsers =>
                <button onClick={addFakeUsers}>Add Fake
                Users</button>
            }
        </Mutation>
        <ul>
            {users.map(user =>
                <UserListItem key={user.githubLogin}
                    name={user.name}
                    avatar={user.avatar} />
            )}
        </ul>
    </div>
```

Так же, как мы отправили запрос в качестве свойства для компонента `Query`, отправим мутацию в компонент `Mutation`. Обратите внимание, что мы применяем свойство `variables`. При этом отправятся необходимые переменные запроса с мутацией. В таком случае устанавливается счетчик, начальное значение которого равно `1`. Это приведет к тому, что мутация будет добавлять одного поддельно-

го пользователя за раз. Компонент `Mutation` применяет функцию `addFakeUsers`, которая будет отправлять мутацию после ее вызова. Когда пользователь нажмет кнопку Add Fake Users (Добавить поддельных пользователей), мутация будет отправлена в наш API.

В настоящее время эти пользователи добавляются в базу данных, но единственный способ увидеть изменения — нажать кнопку Refetch Users (Обновить пользователей). Мы можем указать компоненту `Mutation` выполнить повторную загрузку определенных запросов после завершения мутации, а не ждать, пока пользователи нажмут кнопку:

```
<Mutation mutation={ADD_FAKE_USERS_MUTATION}
    variables={{ count: 1 }}
    refetchQueries={[{ query: ROOT_QUERY }]}>
    {addFakeUsers =>
        <button onClick={addFakeUsers}>Add Fake Users</button>
    }
</Mutation>
```

`refetchQueries` — свойство, которое позволяет указать, какие запросы будут возвращены после отправки мутации. Просто поместите туда список объектов, содержащих запросы. Каждая из операций запроса, найденных в этом списке, будет повторно востребовать данные после завершения мутации.

Авторизация

В главе 5 мы создали мутацию для авторизации пользователя с помощью аккаунта на сайте GitHub. В следующем разделе мы покажем вам, как настроить авторизацию пользователя на стороне клиента.

Процесс авторизации пользователя включает в себя несколько этапов. Вот некоторые общие этапы, указывающие, какую функциональность мы добавили клиенту.

1. *Клиент* перенаправляет пользователя в GitHub с `client_id`.
2. *Пользователь* открывает доступ к информации учетной записи в GitHub для клиентского приложения.

3. *GitHub* перенаправляет обратно на сайт с кодом http://localhost:3000?code=XYZ.
4. *Клиент* отправляет мутацию GraphQL `authUser(code)` с кодом.
5. *API* запрашивает GitHub `access_token` с `client_id`, `client_secret` и `client_code`.
6. *GitHub* отвечает с `access_token`, который может быть задействован в будущих запросах информации.
7. *API* запрашивает информацию пользователя с `access_token`.
8. *GitHub* отвечает с информацией пользователя: `name`, `github_login`, `avatar_url`.
9. *API* распознает мутацию `authUser(code)` с `AuthPayload`, содержащим токен и пользователя.
10. *Клиент* сохраняет токен для отсылки с последующими запросами GraphQL.

Авторизация пользователя

Настало время авторизовать пользователя. Для этого примера мы применяем роутер React, который установим с помощью менеджера npm: `npm install react-router-dom`.

Изменим наш основной компонент `<App />`. Включим компонент `BrowserRouter` и добавим новый компонент `AuthorizedUser`, который можем применять для авторизации пользователей с помощью GitHub:

```
import React from 'react'
import Users from './Users'
import { BrowserRouter } from 'react-router-dom'
import { gql } from 'apollo-boost'
import AuthorizedUser from './AuthorizedUser'

export const ROOT_QUERY = gql`
    query allUsers {
        totalUsers
        allUsers { ...userInfo }
```

```
        me { ...userInfo }
    }

    fragment userInfo on User {
        githubLogin
        name
        avatar
    }
`

const App = () =>
  <BrowserRouter>
    <div>
      <AuthorizedUser />
      <Users />
    </div>
  </BrowserRouter>

export default App
```

Компонент `BrowserRouter` обертывает все остальные компоненты, которые нам нужно визуализировать. Добавим новый компонент `AuthorizedUser`, который будет создан в новом файле. Мы должны получать ошибку, пока не добавим этот компонент.

Мы также изменили `ROOT_QUERY`, чтобы подготовить его к авторизации. Теперь дополнительно запрашиваем поле `me`, которое возвращает информацию о текущем пользователе, когда кто-то вошел в систему. Когда пользователь не авторизован, данное поле просто возвращает `null`. Обратите внимание, что мы добавили фрагмент с именем `userInfo` в документ запроса. Благодаря этому мы можем получить ту же информацию о `User` в двух местах: в поле `me` и поле `allUsers`.

Компонент `AuthorizedUser` должен перенаправить пользователя в GitHub для запроса кода. Этот код должен быть возвращен из GitHub в наше приложение по адресу http://localhost:3000.

В новом файле с именем `AuthorizedUser.js` реализуем данный процесс:

```
import React, { Component } from 'react'
import { withRouter } from 'react-router-dom'

class AuthorizedUser extends Component {
```

```
    state = { signingIn: false }

    componentDidMount() {
        if (window.location.search.match(/code=/)) {
            this.setState({ signingIn: true })
            const code = window.location.search.replace(
            "?code=", "")
            alert(code)
            this.props.history.replace('/')
        }
    }

    requestCode() {
      var clientID = <ID_КЛИЕНТА_GITHUB>
      window.location =
      `https://github.com/login/oauth/authorize?client_
      id=${clientID}&scope=user`
    }

    render() {
        return (
          <button onClick={this.requestCode}
          disabled={this.state.signingIn}>
              Sign In with GitHub
          </button>
        )
    }
}

export default withRouter(AuthorizedUser)
```

Компонент `AuthorizedUser` отвечает за кнопку Sign In with GitHub (Подписаться через GitHub). После ее нажатия пользователь перенаправляется к процессу OAuth GitHub. После авторизации GitHub передаст код обратно в браузер: http://localhost:3000?code=XYZGNARLYSENDABC. Если код найден в строке запроса, компонент анализирует его, исходя из адресной строки, и отображает его в окне предупреждения для пользователя, прежде чем удалить с помощью свойства `history`, которое было отправлено этому компоненту с помощью роутера React.

Вместо того чтобы отправлять пользователю предупреждение с кодом GitHub, нам нужно отправить его в мутацию `githubAuth`:

```
import { Mutation } from 'react-apollo'
import { gql } from 'apollo-boost'
import { ROOT_QUERY } from './App'

const GITHUB_AUTH_MUTATION = gql`
    mutation githubAuth($code:String!) {
        githubAuth(code:$code) { token }
    }
`
```

Вышеупомянутая мутация будет применяться для авторизации текущего пользователя. Нам нужен лишь код. Добавим данную мутацию в метод `render` указанного компонента:

```
render() {
    return (
        <Mutation mutation={GITHUB_AUTH_MUTATION}
            update={this.authorizationComplete}
            refetchQueries={[{ query: ROOT_QUERY }]}>

            {mutation => {
                this.githubAuthMutation = mutation
                return (
                    <button
                        onClick={this.requestCode}
                        disabled={this.state.signingIn}>
                        Sign In with GitHub
                    </button>
                )
           }}

       </Mutation>
    )
}
```

Компонент `Mutation` привязан к `GITHUB_AUTH_MUTATION`. По завершении он вызовет метод `authorizationComplete` компонента и заново вызовет `ROOT_QUERY`. Функция мутации была добавлена в область действия компонента `AuthorizedUser` с помощью кода

`this.githubAuthMutation = mutation`. Теперь мы можем вызвать эту функцию `this.githubAuthMutation()`, когда будем готовы (когда у нас есть код).

Вместо предупреждения отправим код вместе с мутацией для авторизации текущего пользователя. После авторизации сохраним полученный токен в хранилище `localStorage` и применим свойство `history` роутера для удаления кода из адресной строки:

```
class AuthorizedUser extends Component {

    state = { signingIn: false }

    authorizationComplete = (cache, { data }) => {
        localStorage.setItem('token', data.githubAuth.token)
        this.props.history.replace('/')
        this.setState({ signingIn: false })
    }

    componentDidMount() {
        if (window.location.search.match(/code=/)) {
            this.setState({ signingIn: true })
            const code = window.location.search.replace(
            "?code=", "")
            this.githubAuthMutation({ variables: {code} })
        }
    }

    ...

}
```

Чтобы запустить процесс авторизации, вызовем метод `this.githubAuthMutation()` и добавим `code` в переменные операции. После завершения будет вызван метод `authorizationComplete`. Этому методу передаются данные, выбранные нами в мутации, в том числе `token`. Мы сохраним `token` локально и применим `history` роутера React, чтобы удалить строку запроса кода из адресной строки окна.

На данный момент мы подписаны как текущий пользователь GitHub. На следующем этапе нужно убедиться, что мы отправляем данный токен вместе с каждым запросом в заголовках HTTP.

Идентификация пользователя

Наша следующая задача — добавить токен в заголовок авторизации для каждого запроса. Помните, что сервис `photo-share-api`, который мы создали ранее, будет определять пользователей, которые передают токен авторизации в заголовке. Следует лишь убедиться, что любой токен, сохраненный в `localStorage`, отправляется вместе с каждым запросом к нашему сервису GraphQL.

Изменим файл `src/index.js`. Нам нужно найти строку, в которой мы создаем клиент Apollo, и заменить ее этим кодом:

```
const client = new ApolloClient({
    uri: 'http://localhost:4000/graphql',
    request: operation => {
        operation.setContext(context => ({
            headers: {
                ...context.headers,
                authorization: localStorage.getItem('token')
            }
        }))
    }
})
```

Мы добавили метод запроса в нашу конфигурацию клиента Apollo. Этот метод передает сведения о каждой операции `operation` непосредственно перед отправкой в сервис GraphQL. Здесь мы устанавливаем контекст каждой операции для включения заголовка авторизации, который содержит маркер, сохраненный в локальном хранилище. Не волнуйтесь, что у нас нет сохраненного токена, значение этого заголовка будет просто `null`, и наш сервис будет считать, что пользователь не был авторизован.

Теперь, когда мы добавили токен авторизации в каждый заголовок, поле `my` должно возвращать данные о текущем пользователе. Отобразим эти данные в пользовательском интерфейсе. Найдем код метода `render` в компоненте `AuthorizedUser` и заменим его следующим образом:

```
render() {
    return (
```

```
        <Mutation
            mutation={GITHUB_AUTH_MUTATION}
            update={this.authorizationComplete}
            refetchQueries={[{ query: ROOT_QUERY }]}>
            {mutation => {
                this.githubAuthMutation = mutation
                return (
                    <Me signingIn={this.state.signingIn}
                        requestCode={this.requestCode}
                        logout={() => localStorage.
                        removeItem('token')} />
                )
            }}
        </Mutation>
    )
}
```

Вместо того чтобы отображать кнопку, компонент `Mutation` теперь отображает компонент `Me`. Компонент `Me` будет выводить либо информацию о текущем авторизованном пользователе, либо кнопку авторизации. Он должен будет знать, находится ли пользователь в настоящее время в процессе авторизации. Ему также необходимо получить доступ к методам `requestCode` для компонента `AuthorizedUser`. Наконец, нам нужно предоставить функцию, которая может деавторизовать текущего пользователя. На данный момент мы просто удалим токен из `localStorage`, когда пользователь выйдет из системы. Все эти значения были переданы компоненту `Me` как свойства.

Пришло время создать компонент `Me`. Добавим следующий код выше объявления компонента `AuthorizedUser`:

```
const Me = ({ logout, requestCode, signingIn }) =>
    <Query query={ROOT_QUERY}>
        {({ loading, data }) => data.me ?
            <CurrentUser {...data.me} logout={logout} /> :
            loading ?
                <p>loading... </p> :
                <button
                    onClick={requestCode}
                    disabled={signingIn}>
```

```
                        Sign In with GitHub
                </button>
        }
    </Query>

const CurrentUser = ({ name, avatar, logout }) =>
    <div>
        <img src={avatar} width={48} height={48} alt="" />
        <h1>{name}</h1>
        <button onClick={logout}>logout</button>
    </div>
```

Компонент `Me` отображает компонент `Query` для получения данных о текущем пользователе из `ROOT_QUERY`. Если есть токен, поле `me` в `ROOT_QUERY` не будет равно `null`. Внутри компонента запроса мы проверяем, является ли `data.me` значением `null`. Если в этом поле есть данные, мы отобразим компонент `CurrentUser` и передадим данные о текущем пользователе указанному компоненту в качестве свойств. Код `{... data.me}` применяет оператор распространения для передачи всех полей компоненту `CurrentUser` в качестве отдельных свойств. Кроме того, функция выхода из системы передается компоненту `CurrentUser`. Когда пользователь нажмет кнопку выхода из системы, эта функция будет вызвана и его токен будет удален.

Работа с кэшем

Будучи разработчиками, мы вовлечены в процесс минимизации сетевых запросов. Мы не хотим, чтобы нашим пользователям приходилось делать лишние запросы. Чтобы свести к минимуму количество сетевых запросов, отправляемых нашими приложениями, мы можем настроить Apollo Cache.

Политики выборки

По умолчанию Apollo Client хранит данные в локальной переменной JavaScript. Каждый раз, когда мы создаем клиент, для этого создается кэш. Каждый раз, когда мы отправляем операцию, ответ кэшируется

локально. Свойство `fetchPolicy` сообщает Apollo Client, где искать данные для разрешения операции: либо локальный кэш, либо сетевой запрос. По умолчанию для `fetchPolicy` им является значение `cache-first`. Это означает, что клиент будет искать локально в кэше данные для разрешения операции. Если клиент может разрешить операцию без отправки сетевого запроса, он сделает это. Однако если данные для разрешения запроса не находятся в кэше, клиент отправляет сетевой запрос сервису GraphQL.

Другой тип значения `fetchPolicy` — `cache-only`. Эта политика говорит клиенту, что поиск выполняется только в кэше и никогда не посылает сетевой запрос. Если данные для выполнения запроса в кэше не существуют, будет выдана ошибка.

Откройте файл `src/Users.js` и найдите строку `Query` в компоненте `Users`. Можно изменить политику выбора отдельных запросов, добавив свойство `fetchPolicy`:

```
<Query query={{ query: ROOT_QUERY }} fetchPolicy="cache-only">
```

В настоящее время, если мы присвоим политике `Query` значение `cache-only` и обновим страницу в браузере, мы увидим ошибку, потому что Apollo Client просматривает только кэш на предмет данных, чтобы разрешить наш запрос, и эти данные при запуске приложения отсутствуют. Чтобы устранить такую ошибку, измените политику выборки на `cache-and-network`:

```
<Query query={{ query: ROOT_QUERY }} fetchPolicy="cache-and-network">
```

Приложение теперь работает. Политика `cache-and-network` всегда допускает запрос непосредственно из кэша и дополнительно отправляет сетевой запрос для получения актуальных данных. Если локальный кэш не существует, как это происходит в случае запуска приложения, такая политика будет просто извлекать данные из сети. Другие доступные политики:

- `network-only` — всегда посылает сетевой запрос;
- `no-cache` — всегда посылает сетевой запрос для разрешения данных и не кэширует ответ.

Сохранение кэша

Можно сохранить кэш локально на клиенте. Это откроет возможности политики `cache-first`, поскольку кэш уже будет существовать, когда пользователь вернется к приложению. В таком случае политика `cache-first` немедленно допустит данные из существующего локального кэша и вообще не отправит запрос в сеть.

Чтобы сохранять данные кэша локально, нам нужно установить следующий npm-пакет:

```
npm install apollo-cache-persist
```

Пакет `apollo-cache-persist` содержит функцию, которая расширяет кэш, сохраняя его в локальном хранилище всякий раз, когда он изменяется. Чтобы реализовать работу с кэшем, нам нужно создать собственный объект `cache` и добавить его к объекту `client` при настройке нашего приложения.

Добавим следующий код в файл `src/index.js`:

```
import ApolloClient, { InMemoryCache } from 'apollo-boost'
import { persistCache } from 'apollo-cache-persist'

const cache = new InMemoryCache()
persistCache({
    cache,
    storage: localStorage
})

const client = new ApolloClient({
    cache,

    ...

})
```

Итак, мы создали собственный экземпляр кэша, задействуя конструктор `InMemoryCache`, предоставленный `apollo-boost`. Затем импортировали метод `persistCache` из `apollo-cache-persist`. Используя `InMemoryCache`, мы создаем новый экземпляр кэша и отправляем его методу `persistCache` вместе с местом нахождения `storage`. Мы решили

сохранить кэш в хранилище `localStorage` окна браузера. Это означает, что, как только мы запустим наше приложение, мы увидим значение кэша, сохраненного в хранилище. Можно проверить это, добавив такой синтаксис:

```
console.log(localStorage['apollo-cache-persist'])
```

Следующим шагом будет проверка `localStorage` при запуске, чтобы узнать, есть ли у нас уже сохраненный кэш. Если да, тогда нам нужно инициализировать локальный `cache` со следующими данными перед созданием клиента:

```
const cache = new InMemoryCache()
persistCache({
    cache,
    storage: localStorage
})

if (localStorage['apollo-cache-persist']) {
    let cacheData = JSON.parse(localStorage[
    'apollo-cache-persist'])
    cache.restore(cacheData)
}
```

Теперь наше приложение будет загружать любые кэшированные данные до запуска. Если у нас есть данные, сохраненные под ключом `apollo-cache-persist`, мы будем применять метод `cache.restore(cache Data)`, чтобы добавить их в экземпляр `cache`.

Мы успешно минимизировали количество сетевых запросов к нашему сервису, эффективно используя кэш Apollo Client. В следующем разделе вы узнаете, как записывать данные непосредственно в локальный кэш.

Обновление кэша

Компонент `Query` способен непосредственно считывать данные из кэша. Благодаря этому становится возможной политика непосредственной выборки, например `cache-only`. Мы также можем напрямую взаимодействовать с Apollo Cache. Мы можем считывать

текущие данные из кэша или записывать их непосредственно в кэш. Каждый раз, когда мы меняем данные, хранящиеся в кэше, `response-apollo` обнаруживает изменение и повторно отображает все затронутые компоненты. Все, что нам нужно сделать, — это изменить кэш, и пользовательский интерфейс автоматически обновится, чтобы отобразить изменения.

Данные считываются из Apollo Cache с помощью GraphQL. Вы читаете запросы. Данные записываются в Apollo Cache с применением GraphQL, вы записываете данные в запросы. Рассмотрим `ROOT_QUERY`, код которого находится в файле `src/App.js`:

```
export const ROOT_QUERY = gql`
    query allUsers {
        totalUsers
        allUsers { ...userInfo }
        me { ...userInfo }
    }

    fragment userInfo on User {
        githubLogin
        name
        avatar
    }
`
```

Этот запрос имеет три поля в выборке: `totalUsers`, `allUsers` и `me`. Мы можем считывать любые данные, которые в настоящее время хранятся в нашем кэше, используя метод `cache.readQuery`:

```
let { totalUsers, allUsers, me }  = cache.readQuery({ query:
ROOT_QUERY })
```

В этой строке кода мы получили значения для `totalUsers`, `allUsers` и `me`, которые были сохранены в кэше.

Мы также можем записывать данные непосредственно в поля `totalUsers`, `allUsers` и `me` `ROOT_QUERY` с использованием метода `cache.writeQuery`:

```
cache.writeQuery({
    query: ROOT_QUERY,
    data: {
```

```
        me: null,
        allUsers: [],
        totalUsers: 0
    }
})
```

В этом примере мы очищаем все данные из нашего кэша и возвращаем значения по умолчанию для всех полей `ROOT_QUERY`. Поскольку мы применяем `react-apollo`, такое изменение приведет к обновлению пользовательского интерфейса и очистит весь список пользователей в текущей DOM.

Подходящее расположение для записи данных непосредственно в кэш находится внутри функции `logout`, в компоненте `AuthorizedUser`. В настоящее время эта функция удаляет токен пользователя, но пользовательский интерфейс не обновляется до тех пор, пока не будет нажата кнопка Refetch (Перевыбрать) или браузер не обновится. Чтобы решить проблему, мы удаляем текущего пользователя из кэша напрямую, когда он выходит из системы.

Сначала нам нужно убедиться, что этот компонент имеет доступ к объекту `client` в своих свойствах. Один из самых быстрых способов передать данное свойство — использовать компонент более высокого порядка `withApollo`. Это добавит `client` в свойства компонента `AuthorizedUser`. Поскольку данный компонент уже задействует компонент более высокого порядка `withRouter`, мы будем применять функцию `compose`, чтобы убедиться, что компонент `AuthorizedUser` обернут обоими компонентами более высокого порядка:

```
import { Query, Mutation, withApollo, compose }
  from 'react-apollo'

class AuthorizedUser extends Component {
    ...
}

export default compose(withApollo, withRouter)(AuthorizedUser)
```

Используя функцию `compose`, мы объединяем функции `withApollo` и `withRouter` в одну. Функция `withRouter` добавляет свойство `history` роутера, а `withApollo` добавляет к свойствам Apollo Client.

Это означает, что мы можем получить доступ к Apollo Client в нашем методе `logout` и применить его для удаления сведений о текущем пользователе из кэша:

```
logout = () => {
    localStorage.removeItem('token')
    let data = this.props.client.readQuery({ query: ROOT_QUERY })
    data.me = null
    this.props.client.writeQuery({ query: ROOT_QUERY, data })
}
```

Код выше не только удаляет токен текущего пользователя из `localStorage`, но и очищает поле `me` для текущего пользователя, сохраненного в кэше. Теперь, когда пользователи выходят из системы, они сразу же будут видеть кнопку Sign In with GitHub (Подписаться через GitHub), не обновляя браузер. Эта кнопка отображается только тогда, когда в запросе `ROOT_QUERY` нет никаких значений для `me`.

Еще одно расположение, где мы можем улучшить наше приложение, — работа непосредственно с кэшем в файле `src/Users.js`. Сейчас, когда мы нажимаем кнопку Add Fake User (Добавить поддельного пользователя), мутация отправляется в сервис GraphQL. Компонент `Mutation`, который отображает кнопку Add Fake User (Добавить поддельного пользователя), содержит следующее свойство:

```
refetchQueries={[{ query: ROOT_QUERY }]}
```

Это свойство указывает клиенту отправить дополнительный запрос нашему сервису после завершения мутации. Однако мы уже получаем список новых поддельных пользователей в ответе самой мутации:

```
mutation addFakeUsers($count:Int!) {
    addFakeUsers(count:$count) {
        githubLogin
        name
        avatar
    }
}
```

Поскольку у нас уже есть список новых поддельных пользователей, нет необходимости обращаться на сервер для получения той же информации. Нужно получить этот новый список пользователей

в ответе мутации и добавить его непосредственно в кэш. После изменения кэша пользовательский интерфейс будет следовать изменениям.

Найдите код компонента `Mutation` в файле `Users.js`, который поддерживает мутацию `addFakeUsers`, и замените код `refetchQueries` свойством `update`:

```
<Mutation mutation={ADD_FAKE_USERS_MUTATION}
    variables={{ count: 1 }}
    update={updateUserCache}>
    {addFakeUsers =>
        <button onClick={addFakeUsers}>Add Fake User</button>
    }
</Mutation>
```

Теперь, когда мутация завершена, данные ответа будут отправлены функции `updateUserCache`:

```
const updateUserCache = (cache, { data:{ addFakeUsers } }) => {
    let data = cache.readQuery({ query: ROOT_QUERY })
    data.totalUsers += addFakeUsers.length
    data.allUsers = [
        ...data.allUsers,
        ...addFakeUsers
    ]
    cache.writeQuery({ query: ROOT_QUERY, data })
}
```

Когда компонент `Mutation` вызывает функцию `updateUserCache`, он отправляет `cache` и данные, которые были возвращены в ответе мутации.

Нам нужно добавить поддельных пользователей в текущий кэш, поэтому будем считывать данные, которые уже находятся в кэше, применяя запрос `cache.readQuery({ query: ROOT_QUERY })`, а затем выполним добавление. Так мы увеличим общее количество пользователей: `data.totalUsers += addFakeUsers.length`. Затем мы сопоставим текущий список пользователей с поддельными пользователями, которых получили после мутации. Теперь, когда текущие данные были изменены, их можно записать обратно в кэш с помощью `cache.writeQuery({ query: ROOT_QUERY, data })`. Замена данных в кэше заставит

пользовательский интерфейс обновиться и отобразить нового поддельного пользователя.

На данном этапе мы завершили первую версию пользовательской части нашего приложения. Мы можем перечислить всех пользователей, добавить поддельных пользователей и авторизоваться с помощью аккаунта GitHub. Мы создали приложение GraphQL с полным стеком, применяя Apollo Server и Apollo Client. Компоненты `Query` и `Mutation` — это инструменты, которые мы можем задействовать для быстрого создания клиентов с помощью Apollo Client и React.

В главе 7 мы рассмотрим, как добавить подписки и загрузку файлов в приложение PhotoShare. Мы также обсудим новые инструменты в экосистеме GraphQL, которые вы можете включить в свои проекты.

7 GraphQL в реальном мире

На текущий момент мы разработали схему, построили API GraphQL и реализовали клиент, используя Apollo Client. Мы сделали одну полноценную итерацию GraphQL и изучили, как API GraphQL взаимодействуют с клиентами. Теперь пришло время подготовить наши API и клиенты GraphQL для реальной среды.

Чтобы применить полученные навыки в реальной среде, нужно сделать так, чтобы приложения соответствовали необходимым требованиям. Наши приложения позволяют передавать файлы между клиентом и сервером. Они могут использовать веб-сокеты, чтобы передавать данные в реальном времени нашим клиентам. Современные API безопасны и защищают от вредоносных запросов. Чтобы работать с GraphQL в реальной среде, наши приложения должны соответствовать этим требованиям.

Кроме того, нужно подумать о командах разработки. Возможно, вы работаете с командой полного цикла, но чаще всего команды делятся на фронтенд- и бэкенд-разработчиков. Как все ваши сотрудники могут эффективно работать по разным специализациям в стеке GraphQL?

А как насчет изменения содержимого вашей текущей базы кода? В настоящее время у вас, вероятно, много разных сервисов и API, работающих в реальной среде, и вы, вполне возможно, не имеете ни времени, ни ресурсов для перестройки всего с нуля, используя GraphQL.

В этой главе мы рассмотрим все такие требования и проблемы. Мы начнем с еще двух итераций API PhotoShare. Во-первых, включим подписки и передачу данных в режиме реального времени. Во-вторых, разрешим пользователям публиковать фотографии, реализуя решение

для передачи файлов с помощью GraphQL. Завершив эти итерации в приложении PhotoShare, мы рассмотрим способы защиты нашего API GraphQL для противостояния вредоносным клиентским запросам. Мы завершим данную главу, исследуя способы совместной работы команд для эффективной миграции на GraphQL.

Подписки

Обновления в реальном времени являются важной особенностью современных сетевых и мобильных приложений. Современная технология для передачи данных в реальном времени между сайтами и мобильными приложениями — веб-сокеты. Вы можете использовать протокол WebSocket для открытия дуплексных каналов двусторонней связи через TCP-сокет. Это означает, что веб-страницы и приложения могут отправлять и получать данные по одному соединению. Такая технология позволяет получать обновления с сервера непосредственно на веб-страницу в режиме реального времени.

До текущего момента мы реализовали запросы и мутации GraphQL с применением протокола HTTP. HTTP позволяет отправлять и получать данные между клиентом и сервером, но не помогает подключиться к серверу и отслеживать изменения состояния. До появления веб-сокетов единственным способом отслеживания изменений состояния на сервере была постоянная передача HTTP-запросов на сервер в попытке определить, что изменилось. В главе 6 вы видели, как легко выполнить опрос с помощью тега запроса.

Но если мы хотим полностью использовать преимущества новой сети, GraphQL должен иметь возможность поддерживать передачу данных в реальном времени через веб-сокеты в дополнение к HTTP-запросам. Наилучшее решение — использовать *подписки*. Посмотрим, как мы можем реализовать их в GraphQL.

Работа с подписками

В GraphQL вы используете подписки для отслеживания вашего API на предмет конкретных изменений данных. Apollo Server по умолчанию поддерживает подписки. Он включает пару пакетов npm, которые

обычно применяются для настройки веб-сокетов в приложениях GraphQL: `graphql-subscriptions` и `subscriptions-transport-ws`. Npm-пакет `graphql-subscriptions` обеспечивает реализацию шаблона проектирования «Издатель/подписчик» (Pub/Sub). Pub/Sub необходим для публикации изменений данных, которые могут использовать клиентские подписчики. `subscriptions-transport-ws` — это сервер и клиент WebSocket, который позволяет передавать подписки через веб-сокеты. Apollo Server автоматически включает оба указанных пакета для поддержки подписок.

По умолчанию Apollo Server настраивает протокол WebSocket по адресу ws://localhost:4000. Если вы используете простую конфигурацию сервера, которую мы продемонстрировали в начале главы 5, то она поддерживает веб-сокеты без дополнительной настройки.

Поскольку мы работаем с `apollo-server-express`, нужно выполнить несколько шагов, чтобы подписки работали. Найдите файл `index.js` в папке `photo-share-api` и импортируйте функцию `createServer` из модуля `http`:

```
const { createServer } = require('http')
```

Apollo Server автоматически настраивает поддержку подписки, но для этого ему нужен HTTP-сервер. Мы будем использовать функцию `createServer`. Найдите код в нижней части функции `start`, там, где сервис GraphQL запускается на определенном порте с помощью `app.listen(...)`. Замените этот код следующим:

```
const httpServer = createServer(app)
server.installSubscriptionHandlers(httpServer)

httpServer.listen({ port: 4000 }, () =>
    console.log(`GraphQL Server running at
localhost:4000${server.graphqlPath}`)
)
```

Итак, мы создаем новый объект `httpServer` с помощью экземпляра приложения Express. Объект готов обрабатывать все HTTP-запросы, отправленные ему на основе нашей текущей конфигурации Express. У нас также есть экземпляр сервера, где мы можем добавить поддержку протокола WebSocket. Следующая строка кода, `server.installSubscriptionHandlers(httpServer)`, обеспечивает

работу веб-сокетов. Так Apollo Server добавляет необходимые обработчики для поддержки подписки с помощью веб-сокетов. В дополнение к HTTP-серверу наш бэкенд теперь готов принимать запросы по адресу ws://localhost:4000/graphql.

Теперь, когда у нас есть сервер, который поддерживает подписки, пришло время их реализовать.

Публикация фотографий. Нам нужно знать, когда кто-нибудь из наших пользователей публикует фотографию. Это хороший вариант для подписки. Как и все остальное в GraphQL, для реализации подписки нужно начать со схемы. Добавим тип подписки в схему чуть ниже определения типа `Mutation`:

```
type Subscription {
  newPhoto: Photo!
}
```

Подписка `newPhoto` будет применяться для передачи данных клиенту при добавлении фотографий. Мы отправляем операцию подписки с помощью следующей операции языка запросов GraphQL:

```
subscription {
    newPhoto {
        url
        category
        postedBy {
            githubLogin
            avatar
        }
    }
}
```

Эта подписка будет передавать клиенту сведения о новых фотографиях. Подобно `Query` или `Mutation`, GraphQL позволяет нам запрашивать данные о конкретных полях с помощью выборок. Благодаря данной подписке каждый раз, когда появляется новая фотография, мы получим ее `url` и `category` вместе с `githubLogin` и `avatar` пользователя, разместившего эту фотографию.

Когда подписка отправляется нашему сервису, соединение остается открытым. Отслеживаются изменения данных. Каждая добавленная фотография подталкивает данные к подписчику. Если вы настроили подписку на платформе GraphQL Playground, вы заметите,

что кнопка Play (Воспроизвести) превратилась в красную кнопку Stop (Остановить).

Наличие кнопки Stop (Остановить) означает, что подписка в настоящее время открыта и данные отслеживаются. Когда вы нажмете кнопку Stop (Остановить), подписка будет прекращена. Изменения данных не будут отслеживаться.

Настало время взглянуть на мутацию `postPhoto`: она добавляет новые фотографии в базу данных. Нам нужно опубликовать сведения о новой фотографии в нашей подписке в этой мутации:

```
async postPhoto(root, args, { db, currentUser, pubsub }) {

    if (!currentUser) {
        throw new Error('only an authorized user can post a photo')
    }

    const newPhoto = {
        ...args.input,
        userID: currentUser.githubLogin,
        created: new Date()
    }

    const { insertedIds } =
    await db.collection('photos').insert(newPhoto)
    newPhoto.id = insertedIds[0]

    pubsub.publish('photo-added', { newPhoto })

    return newPhoto

}
```

Данный распознаватель ожидает, что в контекст добавлен экземпляр `pubsub`. Мы сделаем это на следующем шаге. Объект `pubsub` может публиковать события и отправлять данные нашему распознавателю. Он похож на `EventEmitter` Node.js. Вы можете использовать его для публикации событий и отправки данных каждому обработчику, который подписался на событие. Здесь мы публикуем событие `photo-added` сразу после того, как вставляем новую фотографию в базу данных. Сведения о новой фотографии передаются как второй аргумент метода `pubsub.publish`. При этом информация о новой

фотографии передается каждому обработчику, который подписался на события `photo-added`.

Затем добавим распознаватель `Subscription`, который будет применяться для подписки на события `photo-added`:

```
const resolvers = {

  ...

  Subscription: {
    newPhoto: {
      subscribe: (parent, args, { pubsub }) =>
        pubsub.asyncIterator('photo-added')
    }
  }
}
```

Распознаватель `Subscription` является корневым. Он должен быть добавлен непосредственно к объекту распознавателя сразу за распознавателями `Query` и `Mutation`. Внутри распознавателя `Subscription` нам нужно определить распознаватели для каждого поля. Поскольку мы определили поле `newPhoto` в нашей схеме, нам нужно убедиться, что в наших распознавателях присутствует новый распознаватель `newPhoto`.

В отличие от распознавателей `Query` или `Mutation` распознаватели `Subscription` содержат метод подписки. Метод подписки получает `parent`, `args` и `context`, как и любые другие функции распознавателя. Внутри этого метода мы подписываемся на конкретные события. В таком случае мы используем `pubsub.asyncIterator` для подписки на события `photo-added`. Каждый раз, когда событие `photo-added` создается `pubsub`, оно будет проходить через эту новую подписку на фото.

Распознаватели подписки в репозитории

Несколько файлов примеров в репозитории GitHub содержат распознаватели. Вышеприведенный код можно найти в файле resolvers/Subscriptions.js.

Распознаватель `postPhoto` и распознаватель подписки `newPhoto` ожидают, что в контексте будет экземпляр `pubsub`. Изменим контекст,

чтобы включить pubsub. Откройте файл index.js и внесите следующие изменения:

```
const { ApolloServer, PubSub } = require('apollo-server-express')

...

async function start() {

    ...

    const pubsub = new PubSub()
    const server = new ApolloServer({
        typeDefs,
        resolvers,
        context: async ({ req, connection }) => {

            const githubToken = req ?
                req.headers.authorization :
                connection.context.Authorization

            const currentUser = await db
                .collection('users')
                .findOne({ githubToken })

            return { db, currentUser, pubsub }
        }
      })

    ...

}
```

В первую очередь нужно импортировать конструктор PubSub из пакета apollo-server-express. Мы применяем этот конструктор для создания экземпляра pubsub и добавления его в контекст.

Возможно, вы обратили внимание, что мы меняем контекстную функцию. Запросы и мутации будут по-прежнему использовать HTTP. Когда мы отправляем одну из этих операций в сервис GraphQL, аргумент запроса req отправляется обработчику контекста. Однако, когда операция является подпиской, HTTP-запрос отсутствует, в связи с чем аргумент req равен null. Информация для подписок вместо этого передается в тот момент, когда клиент подключается к веб-сокету. В таком случае будет отправлен аргумент

`connection` веб-сокета вместо контекстной функции. Когда у нас будет подписка, нам придется передавать данные авторизации через контекст подключения, а не заголовки HTTP-запросов.

Теперь мы готовы опробовать нашу новую подписку. Откройте платформу Playground и запустите подписку:

```
subscription {
    newPhoto {
        name
        url
        postedBy {
            name
        }
    }
}
```

Как только подписка будет запущена, откройте новую вкладку на платформе Playground и запустите мутацию `postPhoto` (рис. 7.1). Каждый раз, когда вы запускаете эту мутацию, вы будете видеть сведения о новых фотографиях, отправленные по подписке.

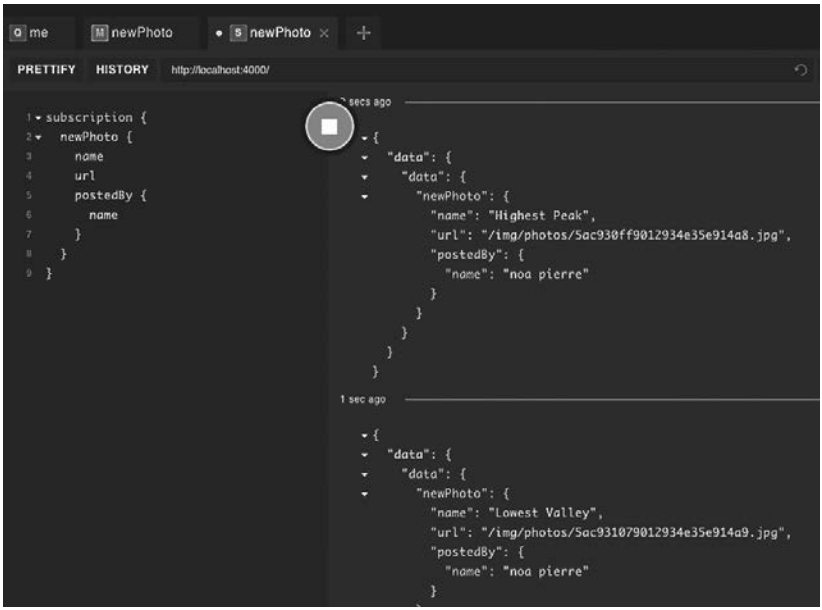

Рис. 7.1. Подписка newPhoto на платформе Playground

Задача: подписка newUser

Можете ли вы реализовать подписку `newUser`? Когда новые пользователи добавляются в базу данных через `githubLogin` или мутацию `addFakeUsers`, можете ли вы опубликовать событие `new-user` для подписки?

Подсказка: при обработке мутации `addFakeUsers` вам может потребоваться опубликовать событие несколько раз, по одному разу для каждого добавленного пользователя.

Если вы не знаете, как решить задачу, то можете найти ответ в репозитории по адресу https://github.com/MoonHighway/learning-graphql/tree/master/chapter-07.

Управление подписками

Предполагая, что вы запустили предыдущую подписку, сервер PhotoShare поддерживает подписки для `Photos` и `Users`. В следующем разделе мы оформляем подписку `newUser` и сразу же выводим на страницу новых пользователей. Прежде чем мы начнем, нужно настроить Apollo Client для обработки подписки.

Добавление WebSocketLink

Подписки реализуются через веб-сокеты. Чтобы задействовать веб-сокеты на сервере, нужно установить несколько дополнительных пакетов:

```
npm install apollo-link-ws apollo-utilities subscription-
transport-ws
```

Теперь необходимо добавить ссылку на веб-сокет в конфигурацию клиента Apollo. Откройте файл `src/index.js` в проекте `photo-share-client` и добавьте следующие команды импорта:

```
import {
    InMemoryCache,
    HttpLink,
    ApolloLink,
    ApolloClient,
    split
} from 'apollo-boost'
import { WebSocketLink } from 'apollo-link-ws'
import { getMainDefinition } from 'apollo-utilities'
```

Обратите внимание, что мы импортировали `split` из `apollo-boost`. Мы будем использовать его для разделения операций GraphQL между HTTP-запросами и веб-сокетами. Если операция является мутацией или запросом, Apollo Client отправит HTTP-запрос. Если операция является подпиской, клиент будет подключаться к веб-сокету.

За кадром Apollo Client сетевые запросы управляются с помощью объекта `ApolloLink`. В текущем приложении он отвечает за отправку HTTP-запросов сервису GraphQL. Каждый раз, когда мы отправляем операцию с Apollo Client, эта операция передается в Apollo Link для обработки сетевого запроса. Мы также можем использовать Apollo Link для работы с сетью через веб-сокеты.

Нам нужно настроить два типа ссылок для поддержки веб-сокетов — `HttpLink` и `WebsocketLink`:

```
const httpLink = new HttpLink({ uri: 'http://localhost:4000/
graphql' })
const wsLink = new WebSocketLink({
    uri: `ws://localhost:4000/graphql`,
    options: { reconnect: true }
})
```

Объект `httpLink` может использоваться для отправки HTTP-запросов по сети на адрес http://localhost:4000/graphql, а `wsLink` можно применять для подключения к ws://localhost:4000/graphql и получения данных через веб-сокеты.

Ссылки являются составными. Это означает, что они могут быть объединены друг с другом для создания пользовательских конвейеров для наших операций GraphQL. Помимо возможности отправить операцию в один `ApolloLink`, мы можем отправить ее через цепочку многоразовых ссылок, где каждая ссылка в цепочке может управлять операцией до того, как она достигнет последней ссылки в цепочке, которая обрабатывает запрос и возвращает результат.

Создадим цепочку ссылок с `httpLink`, добавив пользовательский Apollo Link, который отвечает за добавление заголовка авторизации к операции:

```
const authLink = new ApolloLink((operation, forward) => {
    operation.setContext(context => ({
        headers: {
```

```
            ...context.headers,
            authorization: localStorage.getItem('token')
        }
    }))
    return forward(operation)
})

const httpAuthLink = authLink.concat(httpLink)
```

Объект `httpLink` конкатенируется с `authLink` для обработки авторизации пользователя для HTTP-запросов. Имейте в виду, что функция `.concat` — это не та же функция из языка JavaScript, которая объединяет массивы. Это специальная функция, которая объединяет Apollo Links. После конкатенации мы назвали ссылку `httpAuthLink`, чтобы более четко описать поведение. Когда операция отправляется по данной ссылке, она сначала передается в `authLink`, где заголовок авторизации добавляется в операцию до того, как она будет перенаправлена на объект `httpLink` для обработки сетевого запроса. Если вы знакомы с промежуточным программным обеспечением Express или Redux, процесс вам должен быть знаком.

Теперь нам нужно указать клиенту, какую ссылку использовать. Здесь пригодится функция `split`. Она возвращает одну из двух Apollo Links на основе предиката. Первым аргументом функции `split` является предикат. Предикат — это функция, которая возвращает `true` или `false`. Второй аргумент функции `split` представляет ссылку, которую нужно вернуть, когда предикат возвращает `true`, а третий аргумент представляет ссылку для возврата, когда предикат возвращает `false`.

Реализуем ссылку `split`, которая проверяет, является ли наша операция подписками. Если это подписка, мы будем использовать `wsLink` для работы с сетью, иначе мы будем применять `httpLink`:

```
const link = split(
     ({ query }) => {
        const { kind, operation } = getMainDefinition(query)
        return kind === 'OperationDefinition' && operation ===
           'subscription' },
    wsLink,
    httpAuthLink
)
```

Первый аргумент — функция предиката. Она проверит запрос операции АСД, используя функцию `getMainDefinition`. Если эта операция является подпиской, тогда наш предикат вернет `true`. Когда предикат возвращает `true`, ссылка вернет `wsLink`. Когда предикат возвращает `false`, ссылка вернет `httpAuthLink`.

Наконец, нам нужно изменить конфигурацию Apollo Client, чтобы применить наши пользовательские ссылки, передав `link` и `cache`:

```
const client = new ApolloClient({ cache, link })
```

Теперь наш клиент готов обрабатывать подписки. В следующем разделе мы отправим нашу первую операцию подписки с помощью Apollo Client.

В ожидании новых пользователей

На клиенте мы можем отслеживать новых пользователей, создав константу `LISTEN_FOR_USERS`. Она содержит строку с нашей подпиской, которая вернет новый `githubLogin` пользователя, `name` и `avatar`:

```
const LISTEN_FOR_USERS = gql`
    subscription {
        newUser {
            githubLogin
            name
            avatar
        }
    }
`
```

Затем мы можем задействовать компонент `<Subscription />` для отслеживания новых пользователей:

```
<Subscription subscription={LISTEN_FOR_USERS}>
    {({ data, loading }) => loading ?
        <p>loading a new user...</p> :
        <div>
            <img src={data.newUser.avatar} alt="" />
            <h2>{data.newUser.name}</h2>
        </div>
</Subscription>
```

Как вы можете здесь видеть, компонент `<Subscription />` работает как компоненты `<Mutation />` или `<Query />`. Вы отправляете ему подписку, и, когда новый пользователь получен, его данные передаются функции. Проблема с применением этого компонента в нашем приложении заключается в том, что подписка `newUser` передает одного нового пользователя за раз. Таким образом, предыдущий компонент покажет только последнего нового пользователя, который был создан.

Нам нужно отслеживать новых пользователей. Если клиент PhotoShare запускается и у нас есть новый пользователь, мы добавляем его в текущий локальный кэш.

Когда кэш обновляется, пользовательский интерфейс тоже обновляется, поэтому нет необходимости что-либо менять в нем для новых пользователей.

Изменим код компонента `App`. Сначала преобразуем его в компонент `class`, чтобы можно было применять жизненный цикл компонента React. Когда компонент монтируется, мы начинаем отслеживать новых пользователей через нашу подписку. Когда компонент `App` размонтируется, мы прекращаем отслеживание, вызывая метод отмены подписки:

```
import { withApollo } from 'react-apollo'

...

class App extends Component {

    componentDidMount() {
        let { client } = this.props
        this.listenForUsers = client
            .subscribe({ query: LISTEN_FOR_USERS })
            .subscribe(({ data:{ newUser } }) => {
                const data = client.readQuery({ query: ROOT_QUERY })
                data.totalUsers += 1
                data.allUsers = [
                    ...data.allUsers,
                    newUser
                ]
```

```
                client.writeQuery({ query: ROOT_QUERY, data })
            })
    }

    componentWillUnmount() {
        this.listenForUsers.unsubscribe()
    }

    render() {
        ...
    }
}

export default withApollo(App)
```

Когда мы экспортируем компонент `<App />`, мы используем функцию `withApollo` для передачи клиента в App посредством свойств. Когда компонент примонтируется, мы будем применять клиент для отслеживания новых пользователей. Когда компонент размонтируется, мы прекращаем подписку, задействуя метод `unsubscribe`.

Подписка создается с помощью `client.subscribe().subscribe()`. Первой функцией `subscribe` является метод клиента Apollo, который используется для отправки операции подписки на наш сервис. Он возвращает объект наблюдателя. Вторая функция `subscribe` — это метод объекта наблюдателя. Он применяется для подписки обработчиков на наблюдателя. Обработчики вызываются каждый раз, когда подписка передает данные клиенту. В приведенном выше коде мы добавили обработчик, который захватывает информацию о каждом новом пользователе и добавляет ее непосредственно в Apollo Cache с помощью `writeQuery`.

Теперь, когда добавляются новые пользователи, они мгновенно попадают в наш локальный кэш, который немедленно обновляет пользовательский интерфейс. Поскольку подписка добавляет каждого нового пользователя в список в реальном времени, больше не нужно обновлять локальный кэш из файла `src/Users.js`. Внутри этого файла вы должны удалить функцию `updateUserCache`, а также свойство `update` мутации. Вы можете найти завершенную версию компонента приложения на веб-странице github.com/MoonHighway/learning-graphql/tree/master/chapter-07/photo-share-client.

Выгрузка файлов

Последний шаг к созданию нашего приложения PhotoShare — фактически загрузка фотографии. Для загрузки файла с помощью GraphQL нам необходимо изменить API и клиента, чтобы они могли обрабатывать `multipart/form-data`, тип кодировки, который требуется для передачи файла с телом POST через Интернет. Мы собираемся сделать еще один шаг, который позволит нам передать файл как аргумент GraphQL, чтобы сам файл можно было обрабатывать непосредственно внутри распознавателя.

Чтобы помочь с этой реализацией, мы собираемся использовать два npm-пакета: `apollo-upload-client` и `apollo-upload-server`. Оба указанных пакета предназначены для передачи файлов из браузера посредством протокола HTTP. `apollo-upload-client` будет отвечать за захват файла в браузере и передачу его на сервер с помощью операции. `apollo-upload-server` предназначен для обработки файлов, переданных на сервер от `apollo-upload-client`. Пакет `apollo-upload-server` захватывает файл и сопоставляет его соответствующему аргументу запроса перед отправкой его в распознаватель в качестве аргумента.

Обработка выгрузок на сервере

Apollo Server автоматически задействует сервер `apollo-upload-server`. Нет необходимости устанавливать этот npm-пакет в проекте API, потому что он уже существует и работает. API GraphQL должен быть готов принять загруженный файл. На сервере Apollo предоставляется пользовательский скалярный тип `Upload`. Его можно применять для захвата `stream` файла, `mimetype` и `encoding` загруженного файла.

Мы начнем со схемы, добавив пользовательский скаляр к нашим определениям типов. В файле схемы мы добавим скаляр `Upload`:

```
scalar Upload

input PostPhotoInput {
  name: String!
```

```
  category: Photo_Category = PORTRAIT
  description: String,
  file: Upload!
}
```

Тип `Upload` позволит нам передать содержимое файла с помощью нашего типа ввода `PostPhotoInput`. Это означает, что мы получим сам файл в распознавателе. Тип `Upload` содержит информацию о файле, включая `stream` загрузки, который мы можем использовать для сохранения файла. Применим этот `stream` в мутации `postPhoto`. Добавьте следующий код в конец мутации `postPhoto` в файле `resolvers/Mutation.js`:

```
const { uploadStream } = require('../lib')
const path = require('path')

...

async postPhoto(root, args, { db, user, pubsub }) => {

    ...

    var toPath = path.join(
        __dirname, '..', 'assets', 'photos', `${photo.id}.jpg`
    )

    await { stream } = args.input.file
    await uploadFile(input.file, toPath)

    pubsub.publish('photo-added', { newPhoto: photo })

    return photo
}
```

В этом примере функция `uploadStream` вернет промис, который будет разрешен по завершении загрузки. Аргумент `file` содержит поток загрузки, который может быть отправлен в `writeStream` и сохранен локально в каталоге `assets/photos`. Каждая новая фотография будет именоваться на основе ее уникального идентификатора. Для краткости в данном примере мы обрабатываем только изображения в формате JPEG.

Если нам нужно обрабатывать файлы от одного и того же API, придется добавить специальное промежуточное программное обеспечение в приложение Express, которое позволит обрабатывать статичные изображения в формате JPEG. В файле `index.js`, где мы настраивали наш Apollo Server, мы можем добавить промежуточное программное обеспечение `express.static`, которое позволит обслуживать локальные статичные файлы по маршруту:

```
const path = require('path')

...
app.use(
    '/img/photos',
    express.static(path.join(__dirname, 'assets', 'photos'))
)
```

Этот фрагмент кода обрабатывает статичные файлы в каталогах от `assets/photos` до `/img/photos` для HTTP-запросов.

Теперь наш сервер готов и способен обрабатывать закачки фотографий. Пришло время перейти на сторону клиента, где мы создадим форму для управления загрузкой фотографий.

Использование файловой службы

В реальном приложении Node.js вы обычно сохраняете загружаемые пользователями файлы в облачное хранилище. Предыдущий пример использует функцию uploadFile для загрузки файла в локальный каталог, что ограничивает масштабируемость этого демонстрационного приложения. Такие сервисы, как AWS, Google Cloud или Cloudinary, могут поддерживать большие объемы загрузки файлов из распределенных приложений.

Публикация новых фотографий с помощью Apollo Client

Теперь обработаем фотографии на клиенте. Сначала, чтобы отобразить фотографии, нужно добавить поле `allPhotos` в наш запрос `ROOT_QUERY`. Измените следующий запрос в файле `src/App.js`:

```
export const ROOT_QUERY = gql`
    query allUsers {
```

```
        totalUsers
        totalPhotos
        allUsers { ...userInfo }
        me { ...userInfo }
        allPhotos {
            id
            name
            url
        }
    }

    fragment userInfo on User {
        githubLogin
        name
        avatar
    }
`
```

Теперь, когда сайт загружается, мы получаем значения `id`, `name` и `url` каждой фотографии, сохраненной в базе данных. Мы можем использовать эту информацию для отображения фотографий. Создадим компонент `Photos`, который будет применяться для отображения каждой фотографии:

```
import React from 'react'
import { Query } from 'react-apollo'
import { ROOT_QUERY } from './App'

const Photos = () =>
    <Query query={ALL_PHOTOS_QUERY}>
        {({loading, data}) => loading ?
            <p>loading...</p> :
            data.allPhotos.map(photo =>
                <img
                    key={photo.id}
                    src={photo.url}
                    alt={photo.name}
                    width={350} />

            )
        }
    </Query>

export default Photos
```

Напомним, что компонент `Query` принимает `ROOT_QUERY` как свойство. Затем мы используем шаблон свойства визуализации для отображения всех фотографий при завершении загрузки. Для каждой фотографии в массиве `data.allPhotos` мы добавим новый элемент `img` с метаданными, которые выбираем из каждого объекта фотографии, включая `photo.url` и `photo.name`.

Когда мы добавим этот код в компонент `App`, наши фотографии будут отображаться. Но сначала создадим еще один компонент — `PostPhoto`, который будет содержать форму:

```
import React, { Component } from 'react'

export default class PostPhoto extends Component {

    state = {
        name: '',
        description: '',
        category: 'PORTRAIT',
        file: ''
    }

    postPhoto = (mutation) => {
        console.log('todo: post photo')
        console.log(this.state)
    }

    render() {
        return (
            <form onSubmit={e => e.preventDefault()}
                style={{
                    display: 'flex',
                    flexDirection: 'column',
                    justifyContent: 'flex-start',
                    alignItems: 'flex-start'
                }}>

                <h1>Post a Photo</h1>

                <input type="text"
                    style={{ margin: '10px' }}
```

```
        placeholder="photo name..."
        value={this.state.name}
        onChange={({target}) =>
            this.setState({ name: target.value })} />

    <textarea type="text"
        style={{ margin: '10px' }}
        placeholder="photo description..."
        value={this.state.description}
        onChange={({target}) =>
            this.setState({ description:
            target.value })} />

    <select value={this.state.category}
        style={{ margin: '10px' }}
        onChange={({target}) =>
            this.setState({ category:
            target.value })}>
        <option value="PORTRAIT">PORTRAIT</option>
        <option value="LANDSCAPE">LANDSCAPE</option>
        <option value="ACTION">ACTION</option>
        <option value="GRAPHIC">GRAPHIC</option>
    </select>

    <input type="file"
        style={{ margin: '10px' }}
        accept="image/jpeg"
        onChange={({target}) =>
            this.setState({
                file: target.files &&
                target.files.length ?
                    target.files[0] :
                    ''
        })} />

    <div style={{ margin: '10px' }}>
        <button onClick={() => this.postPhoto()}>
            Post Photo
        </button>
        <button onClick={() =>
        this.props.history.goBack()}>
```

```
                    Cancel
                </button>
            </div>

        </form>
    )
  }
}
```

Компонент `PostPhoto` — просто форма. В ней используются элементы формы для ввода `name`, `description`, `category` и самого `file`. В React мы называем такую форму управляемой, потому что каждый элемент ввода связан с переменной состояния. Каждый раз, когда изменяется значение ввода, состояние компонента `PostPhoto` также изменяется.

Мы отправляем фотографии, нажимая кнопку Post Photo (Опубликовать фото). Элемент формы для ввода файла принимает формат JPEG и устанавливает состояние для `file`. Это поле состояния представляет собой фактический файл, а не только текст. Для краткости мы не добавили проверку формы в данном компоненте.

Пришло время добавить наши новые компоненты в компонент `App`. Когда мы это сделаем, мы убедимся, что домашний маршрут отображает наши `Users` и `Photos`. Мы также добавим маршрут `/newPhoto`, который можно использовать для отображения формы.

```
import React, { Fragment } from 'react'
import { Switch, Route, BrowserRouter } from 'react-router-dom'
import Users from './Users'
import Photos from './Photos'
import PostPhoto from './PostPhoto'
import AuthorizedUser from './AuthorizedUser'

const App = () =>
    <BrowserRouter>
        <Switch>
            <Route
                exact
                path="/"
                component={() =>
                    <Fragment>
                        <AuthorizedUser />
```

```
                        <Users />
                        <Photos />
                    </Fragment>
                } />
            <Route path="/newPhoto" component={PostPhoto} />
            <Route component={({ location }) =>
                <h1>"{location.pathname}" not found</h1>
            } />
        </Switch>
    </BrowserRouter>

export default App
```

Компонент `<Switch>` позволяет отображать один маршрут за раз. Если элемент `url` содержит домашний маршрут, с помощью `/` мы отображаем компонент, включающий компоненты `AuthorizedUser`, `Users` и `Photos`. `Fragment` применяется в React, когда нужно отображать компоненты того же уровня без необходимости их обертывания в дополнительный элемент `div`. Если указан маршрут `/newPhoto`, мы отобразим форму для загрузки новой фотографии. И если маршрут не будет распознан, выводится элемент `h1`, позволяющий пользователю узнать, что мы не можем найти маршрут, который он предоставил.

Только авторизованные пользователи могут публиковать фотографии, поэтому мы добавим элемент `NavLink` со значением `Post Photo` к компоненту `AuthorizedUser`. Нажатие данной кнопки приведет к визуализации `PostPhoto`.

```
import { withRouter, NavLink } from 'react-router-dom'

...

class AuthorizedUser extends Component {

    ...

    render() {
        return (
            <Query query={ME_QUERY}>
                {({ loading, data }) => data.me ?
                    <div>
```

```
                <img
                    src={data.me.avatar_url}
                    width={48}
                    height={48}
                    alt="" />
                <h1>{data.me.name}</h1>
                <button onClick={this.logout}>
                logout</button>
                <NavLink to="/newPhoto">Post
                Photo</NavLink>
            </div> :

...
```

Здесь мы импортируем компонент `<NavLink>`. При щелчке на ссылке Post Photo (Опубликовать фото) пользователь будет отправлен к маршруту `/newPhoto`.

На данном этапе приложение должно работать. Пользователь может перемещаться между экранами, а при публикации фотографии мы должны увидеть необходимые входные данные, выведенные в оболочке командной строки. Настало время взять эти данные, включая файл, и отправить их с помощью мутации.

В первую очередь установим `apollo-upload-client`:

```
npm install apollo-upload-client
```

Мы заменим текущую HTTP-ссылку той, которая предоставляется пакетом `apollo-upload-client`. Эта ссылка будет поддерживать запросы `multipart/form-data`, содержащие файлы загрузки. Чтобы создать данную ссылку, мы используем функцию `createUploadLink`:

```
import { createUploadLink } from 'apollo-upload-client'

...

const httpLink = createUploadLink({
    uri: 'http://localhost:4000/graphql'
})
```

Мы заменили старую HTTP-ссылку новой, вызванной с помощью функции `createUploadLink`. У нее есть API-маршрут, обозначенный как `uri`.

Пришло время добавить мутацию postPhoto в форму PostPhoto:

```
import React, { Component } from 'react'
import { Mutation } from 'react-apollo'
import { gql } from 'apollo-boost'
import { ROOT_QUERY } from './App'

const POST_PHOTO_MUTATION = gql`
    mutation postPhoto($input: PostPhotoInput!) {
        postPhoto(input:$input) {
            id
            name
            url
        }
    }
`
```

POST_PHOTO_MUTATION — это наша мутация, разобранная как АСД и готовая к отправке на сервер. Мы импортируем ALL_PHOTOS_QUERY, потому что нам нужно будет использовать его, когда придет время обновить локальный кэш с новой фотографией, которая будет возвращена мутацией.

Чтобы добавить мутацию, мы инкапсулируем элемент кнопки Post Photo (Опубликовать фото) в компонент Mutation:

```
<div style={{ margin: '10px' }}>
    <Mutation mutation={POST_PHOTO_MUTATION}
        update={updatePhotos}>
        {mutation =>
            <button onClick={() => this.postPhoto(mutation)}>
                Post Photo
            </button>
        }
    </Mutation>
    <button onClick={() => this.props.history.goBack()}>
        Cancel
    </button>
</div>
```

Компонент Mutation передает мутацию как функцию. Нажимая кнопку, мы передаем функцию мутации в postPhoto, чтобы ее можно было использовать для изменения фотоданных. Как только мутация

будет завершена, вызывается функция `updatePhotos` для обновления локального кэша.

Следующим шагом фактически отправим мутацию:

```
postPhoto = async (mutation) => {
    await mutation({
        variables: {
            input: this.state
        }
    }).catch(console.error)
    this.props.history.replace('/')
}
```

Эта функция мутации возвращает промис. По завершении мы будем применять роутер React, чтобы отправить пользователя обратно на домашнюю страницу, заменяя текущий маршрут с помощью свойства `history`. Когда мутация завершена, нам нужно захватить данные, возвращенные из нее, для обновления содержимого локального кэша:

```
const updatePhotos = (cache, { data:{ postPhoto } }) => {
    var data = cache.readQuery({ query: ALL_PHOTOS_QUERY })
    data.allPhotos = [
        postPhoto,
        ...allPhotos
    ]
    cache.writeQuery({ query: ALL_PHOTOS_QUERY, data })
}
```

Метод `updatePhotos` поддерживает обновление кэша. Мы будем считывать фотографии из кэша, используя запрос `ROOT_QUERY`. Затем добавим новую фотографию в кэш с помощью метода `writeQuery`. Так мы сможем убедиться, что локальные данные синхронизированы.

На этом этапе вы готовы публиковать новые фотографии. Продолжайте и экспериментируйте.

Мы внимательно рассмотрели, как запросы, мутации и подписки обрабатываются на стороне клиента. Когда вы применяете React Apollo, можете прибегнуть к преимуществам компонентов `<Query>`, `<Mutation>` и `<Subscription>` — они помогут подключить данные из вашего сервиса GraphQL к вашему пользовательскому интерфейсу.

Теперь, когда приложение работает, для безопасности мы добавим еще один уровень.

Безопасность

Ваш сервис GraphQL обеспечивает большую свободу и гибкость для ваших клиентов. Им предоставлена гибкость для запроса данных из нескольких источников в одном запросе. Они также могут запрашивать в одном запросе большое количество связанных или соединенных данных, а также без проверки запросить слишком много данных у вашего сервиса. Мало того, что загрузка большими запросами влияет на производительность сервера, она также может привести к полному отказу вашего сервиса. Одни клиенты могут сделать это невольно или непреднамеренно, тогда как другие — умышленно. В любом случае вам необходимо принять некоторые меры предосторожности и контролировать производительность вашего сервера для защиты от больших или злонамеренных запросов.

В этом разделе мы рассмотрим некоторые из доступных вариантов улучшения безопасности вашего сервиса GraphQL.

Тайм-ауты запроса

Тайм-аут запроса представляет собой первую защиту от больших или злонамеренных запросов. Тайм-аут запроса позволяет обрабатывать каждый запрос только определенное количество времени. Это означает, что запросы вашего сервиса должны быть завершены в течение конкретного промежутка времени. Тайм-ауты запросов используются не только для сервисов GraphQL, но и для всех видов услуг и процессов в Интернете. Возможно, вы уже внедрили такие тайм-ауты для вашего API передачи репрезентативного состояния (REST) для защиты от длинных запросов со слишком большим количеством данных POST.

Вы можете добавить общий тайм-аут запроса на сервер Express, установив ключ `timeout`. Ниже мы добавили тайм-аут 5 секунд, чтобы избежать нежелательных запросов:

```
const httpServer = createServer(app)
server.installSubscriptionHandlers(httpServer)

httpServer.timeout = 5000
```

Кроме того, вы можете установить тайм-ауты для общих запросов или отдельных распознавателей. Решение для реализации тайм-аутов для запросов или распознавателей заключается в том, чтобы сохранить время начала для каждого запроса или распознавателя и проверить его в сравнении с предпочтительным для вас тайм-аутом. Вы можете записать время начала каждого запроса в контексте:

```
const context = async ({ request }) => {

        ...

        return {
            ...
            timestamp: performance.now()
        }

    }
```

Теперь каждый из распознавателей узнает, когда начался запрос, и может выдать ошибку, если запрос занимает слишком много времени.

Ограничения данных

Еще одна простая защита, которую вы можете противопоставить большим или злонамеренным запросам, заключается в ограничении объема данных, которые могут быть возвращены каждым запросом. Вы можете вернуть определенное количество записей или страницу данных, разрешив вашим запросам указывать, сколько записей нужно вернуть.

Например, вспомните, что в главе 4 мы разработали схему, которая могла бы обрабатывать пагинацию данных. Но что, если клиент запросил чрезвычайно большую страницу данных? Ниже представлен пример того, как клиент делает это:

```
query allPhotos {
  allPhotos(first=99999) {
    name
    url
    postedBy {
        name
        avatar
    }
}
```

Вы можете защититься от больших запросов таких типов, просто установив лимит для страницы данных. Например, установить предел 100 фотографий на запрос на вашем сервере GraphQL. Этот лимит может быть применен в распознавателе запроса при проверке аргумента:

```
allPhotos: (root, data, context) {
    if (data.first > 100) {
        throw new Error('Only 100 photos can be requested at
        a time')
    }
}
```

Если у вас есть большое количество записей, которые могут быть запрошены, всегда рекомендуется внедрять функцию пагинации данных. Вы можете реализовать ее, просто указав количество записей, которые должны быть возвращены после запроса.

Ограничение глубины запроса

Одним из преимуществ GraphQL, предоставляемым клиенту, является возможность запроса связанных данных. Например, в нашем API для фотографий мы можем написать запрос, который предоставит информацию о фотографии, о том, кто разместил ее и все остальные фотографии, опубликованные этим фотографом, — все в одном запросе:

```
query getPhoto($id:ID!) {
    Photo(id:$id) {
        name
        url
        postedBy {
            name
            avatar
            postedPhotos {
                name
                url
            }
        }
    }
}
```

Это хорошая функция, которая может улучшить производительность сетевых взаимодействий в ваших приложениях. Мы можем сказать, что предыдущий запрос имеет глубину 3, потому что запрашивает саму фотографию вместе с двумя связанными полями: `postedBy` и `postedPhotos`. Корневой запрос имеет глубину 0, поле `Photo` — глубину 1, поле `postedBy` — 2, а поле `postedPhotos` — 3.

Клиенты могут воспользоваться преимуществами этого качества. Рассмотрим следующий запрос:

```
query getPhoto($id:ID!) {
    Photo(id:$id) {
        name
        url
        postedBy {
            name
            avatar
            postedPhotos {
                name
                url
                taggedUsers {
                    name
                    avatar
                    postedPhotos {
                        name
                        url
                    }
                }
            }
        }
    }
}
```

Мы добавили еще два уровня к глубине данного запроса: `taggedUsers` во всех фотографиях, опубликованных автором оригинального снимка, и `postedPhotos` всех `taggedUsers` во всех фотографиях, опубликованных автором. Это означает, что, если я разместил исходную фотографию, данный запрос также будет разбираться для всех снимков, которые я опубликовал, всех пользователей, отмеченных на этих фотографиях, и всех фотографий, размещенных всеми этими от-

меченными пользователями. Слишком много данных для запроса. Это также предполагает немало работы для ваших распознавателей. Глубина запроса растет экспоненциально и может легко выйти из-под контроля.

Вы можете ограничить глубину запроса для своих сервисов GraphQL, чтобы предотвратить отказ от обслуживания вашего сервиса из-за слишком глубоких запросов. Если бы мы установили ограничение глубины запроса 3, первый запрос был бы в пределах лимита, тогда как второй запрос — нет, потому его глубина запроса равна 5.

Ограничения глубины запроса обычно реализуются путем анализа АСД запроса и определения того, насколько глубоко вложены выборки в этих объектах. Существуют npm-пакеты, такие как `graphql-depth-limit`, которые могут помочь в выполнении данной задачи:

```
npm install graphql-depth-limit
```

После установки пакета вы можете добавить правило проверки в конфигурацию сервера GraphQL с помощью функции `depthLimit`:

```
const depthLimit = require('graphql-depth-limit')

...

const server = new ApolloServer({
    typeDefs,
    resolvers,
    validationRules: [depthLimit(5)],
    context: async({ req, connection }) => {
        ...
    }
})
```

Здесь мы установили лимит глубины запроса равным 10, а это значит, что мы предоставили нашим клиентам возможность писать запросы, которые могут содержать десять выборок. Если запросы глубже, сервер GraphQL предотвращает их выполнение и возвращает ошибку.

Ограничение сложности запроса

Другим показателем, который может помочь вам выявить проблемы с запросами, является *сложность запроса*. Встречаются клиентские запросы, которые могут не требовать слишком большой глубины, но все равно быть нежелательными из-за количества запрашиваемых полей. Рассмотрим такой запрос:

```
query everything($id:ID!) {
  totalUsers
  Photo(id:$id) {
    name
    url
  }
  allUsers {
    id
    name
    avatar
    postedPhotos {
      name
      url
    }
    inPhotos {
      name
      url
      taggedUsers {
        id
      }
    }
  }
}
```

Запрос `everything` не превышает ограничение по глубине запроса, но он по-прежнему довольно затратный из-за количества запрашиваемых полей. Помните, что каждое поле отображается на функцию `resolver`, которая должна быть вызвана.

Сложность запроса состоит из значения сложности каждого поля, которые затем суммируются в общую сложность любого запроса. Вы можете установить общее ограничение, которое определяет максимальную сложность, доступную для любого заданного запроса. При реализации сложности запроса вы можете определить свои за-

тратные распознаватели и дать этим полям более высокую степень сложности.

Доступно несколько пакетов npm, призванных помочь в реализации ограничений сложности запросов. Посмотрим, как мы можем реализовать сложность запросов в нашем сервисе, используя пакет `graphql-validation-complexity`:

```
npm install graphql-validation-complexity
```

Проверка сложности в GraphQL содержит набор стандартных правил для определения сложности запроса. Каждому скалярному полю присваивается значение `1`. Если это поле находится в списке, сложность увеличивается в десять раз.

Например, посмотрим, как функция `graphql-validation-complexity` будет оценивать запрос `everything`:

```
query everything($id:ID!) {
  totalUsers         # Сложность 1
  Photo(id:$id) {
    name             # Сложность 1
    url              # Сложность 1
  }
  allUsers {
    id               # Сложность 10
    name             # Сложность 10
    avatar           # Сложность 10
    postedPhotos {
      name           # Сложность 100
      url            # Сложность 100
    }
    inPhotos {
      name           # Сложность 100
      url            # Сложность 100
      taggedUsers {
        id           # Сложность 1000
      }
    }
  }
}                    # Итоговая сложность 1433
```

По умолчанию функция `graphql-validation-complexity` присваивает каждому полю значение. Она умножает это значение на 10 для

любого списка. В данном примере `totalUsers` представляет собой одно целочисленное поле и ему присваивается сложность `1`. Запрос полей на единственной фотографии имеет то же значение. Обратите внимание, что полям, запрошенным в списке `allUsers`, присваивается значение `10`. Это происходит потому, что они находятся в списке. Каждое поле списка умножается на 10. Таким образом, списку в списке присваивается значение `100`. Поскольку `taggedUsers` — список в списке `inPhotos`, который находится в списке `allUsers`, значения полей `taggedUser` равны 10 ї 10 ї 10, или 1000.

Мы можем предотвратить выполнение этого конкретного запроса, установив общий предел сложности запросов 1000:

```
const { createComplexityLimitRule } =
  require('graphql-validation-complexity')

...

   const options = {

        ...

        validationRules: [
            depthLimit(5),
            createComplexityLimitRule(1000, {
                onCost: cost => console.log('query cost: ', cost)
            })
        ]
   }
```

В этом примере мы устанавливаем максимальный предел сложности 1000 с использованием функции `createComplexityLimitRule` из пакета `graphql-validation-complexity`. Мы также реализовали функцию `onCost`, которая будет вызвана с общей стоимостью каждого запроса, как только он будет рассчитан. Предыдущий запрос не будет разрешен для выполнения в данных условиях, поскольку он превышает максимальную сложность 1000.

Большинство пакетов сложности запросов позволяют вам устанавливать свои собственные правила. Мы могли бы изменить значения сложности, назначенные скалярам, объектам и спискам, с помощью

пакета `graphqlvalidation-complexity`. Возможно также установить пользовательские значения сложности для любого поля, которое мы считаем очень усложненным или дорогостоящим.

Движок Apollo

Не рекомендуется просто реализовывать функции безопасности и надеяться на лучшее. Любая хорошая стратегия безопасности и производительности требует использования показателей. Вам нужен способ мониторить ваш сервис GraphQL, чтобы вы могли определить свои популярные запросы и узнать, где находятся узкие места в производительности.

Вы можете использовать *Apollo Engine* для мониторинга вашего сервиса GraphQL, но это больше, чем просто инструмент мониторинга. Apollo Engine — надежный облачный сервис, который обеспечивает глубокое понимание вашего уровня GraphQL, чтобы вы могли безопасно и с уверенностью управлять сервисом. Он отслеживает операции GraphQL, отправленные вашим сервисам, и предоставляет детализированный актуальный отчет, доступный в Интернете на странице https://engine.apollographql.com, который вы можете использовать для определения наиболее популярных запросов, мониторинга времени выполнения, мониторинга ошибок и поиска узких мест. Он также предоставляет инструменты для управления схемой, включая проверку.

Apollo Engine уже включен в реализацию Apollo Server 2.0. С помощью всего одной строки кода вы можете запускать Engine везде, где работает Apollo Server, в том числе в серверных и пограничных средах. Все, что вам нужно сделать, — включить его, присвоив ключу `engine` значение `true`:

```
const server = new ApolloServer({
   typeDefs,
   resolvers,
   engine: true
 })
```

Следующий шаг — убедиться, что у вас есть переменная среды `ENGINE_API_KEY`, установленная в вашем ключе API Apollo Engine.

Начните с посещения сайта engine.apollographql.com, создания учетной записи и генерации своего ключа.

Чтобы опубликовать приложение в Apollo Engine, вам необходимо установить инструменты CLI (интерфейса командной строки) Apollo:

```
npm install -g apollo
```

После установки вы можете использовать CLI-инструментарий для публикации своего приложения:

```
apollo schema:publish
    --key=<ВАШ_API_КЛЮЧ>
    --endpoint=http://localhost:4000/graphql
```

Не забудьте также добавить свой API-ключ к переменным окружения.

Теперь, когда вы запустите API PhotoShare GraphQL, все операции, отправляемые сервису GraphQL, будут отслеживаться. Вы можете просмотреть отчет о деятельности на сайте Engine. Этот отчет может использоваться для поиска и устранения узких мест. Кроме того, Apollo Engine улучшит производительность и время отклика наших запросов, а также проверит производительность сервиса.

Дальнейшее обучение

Повсюду в этой книге вы встречались с теорией графов; вы писали запросы; вы разрабатывали схемы; вы настраивали серверы GraphQL и исследовали клиентские решения GraphQL. Базис задан, поэтому вы можете использовать то, что нужно для улучшения ваших приложений с помощью GraphQL. В данном разделе мы поговорим о некоторых концепциях и ресурсах, которые помогут вам в работе над будущими приложениями GraphQL.

Инкрементная миграция

Наше приложение PhotoShare — яркий пример проекта Greenfield. Когда вы работаете над своими проектами, у вас может не быть такой роскоши, как разработка с нуля. Гибкость GraphQL позволяет

вам постепенно внедрять его. Нет никакой причины, по которой вам нужно все удалять и начинать использовать только GraphQL. Вы можете плавно начать обновлять свои приложения, ориентируясь на следующие идеи.

- *Считывайте данные из REST в ваших распознавателях.* Вместо того чтобы перестраивать каждую конечную точку REST, задействуйте GraphQL в качестве шлюза и сделайте запрос выборки для этих данных на сервере внутри распознавателя. Ваш сервис также может кэшировать данные, отправленные из REST, чтобы улучшить время ответа на запрос.
- *Или используйте GraphQL-запрос.* Надежные решения для клиентов — это прекрасно, но их реализация поначалу может быть слишком сложной. Для начала просто примените `graphql-request` и сделайте запрос в том же месте, где вы используете `fetch` для API REST. Этот подход поможет вам начать работу, подключившись к GraphQL, и, скорее всего, приведет к комплексному клиентскому решению, когда вы будете готовы оптимизировать производительность. Нет причин, по которым вы не можете получать данные из четырех конечных точек REST и одного сервиса GraphQL в одном приложении. Все сразу не нужно одновременно переносить на GraphQL.
- *Включайте GraphQL в один или два компонента.* Вместо того чтобы перестраивать весь сайт, выберите один компонент или страницу и поместите данные в эту конкретную область с помощью GraphQL. Держите все остальное на своем месте, пока следите за перемещением одного компонента.
- *Не создавайте больше конечных точек REST.* Вместо расширения REST создайте конечную точку GraphQL для нового сервиса или возможности. Вы можете разместить конечную точку GraphQL на том же сервере, что и конечные точки REST. Express не заботит, маршрутизирует он запрос функции REST или распознавателя GraphQL. Каждый раз, когда задача требует новой конечной точки REST, добавьте эту возможность в свой сервис GraphQL.

- *Не поддерживайте текущие конечные точки REST.* В следующий раз, когда появится задача изменить конечную точку REST или создать пользовательскую конечную точку для некоторых данных — не надо этого делать! Вместо этого найдите время, чтобы отделить данную конечную точку и обновить ее до GraphQL. Вы можете постепенно переместить весь API REST таким образом.

Медленный переход к GraphQL позволяет вам сразу же извлечь выгоду из особенностей без потерь, связанных со стартом с нуля. Начните с того, что у вас есть, и вы сможете сделать свой переход на GraphQL гладким и постепенным.

В первую очередь — разработка схемы

Вы на совещании по поводу нового веб-проекта. Участвуют члены различных фронтенд- и бэкенд-команд. Команды начинают кодирование, но без четких рекомендаций проект разрабатывается вне графика и не соответствует первоначальным ожиданиям каждого.

Проблемы с веб-проектами обычно возникают из-за отсутствия взаимосвязи или недопонимания того, что должно быть в итоге получено. Схемы обеспечивают ясность и коммуникацию, поэтому многие проекты первоначально занимаются *разработкой схемы*. Вместо того чтобы увязнуть в деталях реализации конкретных доменов, разрозненные команды могут работать вместе над упрощением схемы и уже потом начинать что-либо строить.

Схемы — это соглашение между командами фронтенд- и бэкенд-разработки и определение всех отношений между данными в приложении. Когда команды подписываются на схему, они могут работать независимо. Работа по схеме дает лучшие результаты, потому что есть ясность в определениях типов. Команды фронтенд-разработки точно знают, какие запросы сделать для загрузки данных в пользовательские интерфейсы. Команды бэкенд-разработки точно знают, что нужно данным и как их обрабатывать. Разработка схемы в первую очередь дает четкий план, и команды могут строить проект с большим пониманием и меньшим стрессом.

Макеты — важная часть разработки схемы как основного этапа. После того как у команды фронтенд-разработки появляется схема, они могут использовать ее для немедленного запуска компонентов. Следующий код подойдет для того, чтобы включить макет с применением функции GraphQL, запущенной по адресу http://localhost:4000:

```
const { ApolloServer } = require('apollo-server')
const { readFileSync } = require('fs')

var typeDefs = readFileSync('./typeDefs.graphql', 'UTF-8')

const server = new ApolloServer({ typeDefs, mocks: true })

server.listen()
```

Предполагая, что вы предоставили файл `typeDefs.graphql`, разработанный на первом этапе, вы можете приступить к разработке компонентов пользовательского интерфейса, которые отправляют операции запроса, мутации и подписки на сервис-макет (mock) GraphQL, в то время как команда бэкенд-разработки реализует реальный сервис.

Макеты работают без дополнительной настройки, предоставляя значения по умолчанию для каждого типа скаляра. Везде, где предполагается, что поле представлено строкой, вы увидите значение `Hello World`.

Вы можете настроить данные, которые возвращает mock-сервер, чтобы они были больше похожи на реальные. Это важная особенность, которая поможет решить задачу стилизации компонентов пользовательского интерфейса:

```
const { ApolloServer, MockList } = require('apollo-server')
const { readFileSync } = require('fs')

const typeDefs = readFileSync('./typeDefs.graphql', 'UTF-8')

const resolvers = {}

const mocks = {
  Query: () => ({
    totalPhotos: () => 42,
    allPhotos: () => new MockList([5, 10]),
```

```
    Photo: () => ({
      name: 'sample photo',
      description: null
    })
  })
}

const server = new ApolloServer({
  typeDefs,
  resolvers,
  mocks
})

server.listen({ port: 4000 }, () =>
  console.log(`Mock Photo Share GraphQL Service`)
)
```

Этот код добавляет mock-поля `totalPhotos` и `allPhotos` вместе с типом `Photo`. Каждый раз, когда мы запрашиваем `totalPhotos`, возвращается значение `42`. Когда мы запрашиваем поле `allPhotos`, мы получаем 5–10 фотографий. Конструктор `MockList` включен в Apollo Server и используется для создания типов списков с определенной длиной. Каждый раз, когда тип `Photo` допускается сервисом, атрибут `name` фотографии определяется как «пример фотографии», а описание равно `null`. Вы можете создавать довольно надежные макеты с помощью таких пакетов, как `faker` или `casual`. Эти npm-пакеты предоставляют всевозможные поддельные данные, которые можно использовать для создания реалистичных макетов.

Чтобы узнать больше о макетировании в Apollo Server, ознакомьтесь с документацией Apollo (www.apollographql.com/docs/apollo-server/v2/features/mocking.html).

События GraphQL

GraphQL подробно обсуждается на следующих конференциях и встречах.

- *Саммит GraphQL* (summit.graphql.com). Конференция, организованная Apollo GraphQL.

- *День GraphQL* (www.graphqlday.org). Практическая конференция разработчиков в Нидерландах.
- *GraphQL Европа* (www.graphql-europe.org). Некоммерческая конференция GraphQL в Европе.
- *GraphQL Финляндия* (graphql-finland.fi). Организованная сообществом конференция GraphQL в Хельсинки, Финляндия.

Вы также встретите упоминания GraphQL практически на любой конференции по разработке, особенно посвященной JavaScript.

Если вы ищете мероприятия рядом с вами, то имейте в виду, что встречи по теме GraphQL проводятся в разных городах по всему миру (bit.ly/2lnBMB0). Если рядом с вами такого нет, можете стать первыми, кто создаст местную группу!

Сообщество

Язык GraphQL популярен, потому что он замечательный. Он также популярен благодаря активной поддержке сообщества GraphQL. Сообщество довольно приветливое, и есть разные способы влиться в него и оставаться в курсе последних изменений.

Знания, которые вы приобрели в GraphQL, станут хорошей основой для изучения других библиотек и инструментов. Если вы хотите сделать следующие шаги, чтобы расширить свои знания, вот еще несколько тем, к которым нужно присмотреться.

- *Сшивание схемы*. Сшивание схемы позволяет вам создать единую схему GraphQL из нескольких API GraphQL. Apollo предоставляет отличные инструменты для композиции удаленных схем. Узнайте больше об этом в документации Apollo (http://bit.ly/2KcibP6).
- *Prisma*. Всюду в книге мы использовали GraphQL Playground и GraphQL Request: два инструмента команды Prisma. Prisma — это инструмент, который превращает вашу базу данных в API GraphQL, независимо от того, какую именно базу данных вы применяете. Хотя API GraphQL находится между клиентом и базой данных, Prisma стоит между API GraphQL и базой данных. Prisma — проект с открытым исходным кодом, поэтому вы можете

развернуть свой сервис Prisma в реальной среде с помощью любого облачного провайдера.

Команда также выпустила инструмент под названием Prisma Cloud — хостинговую платформу для сервисов Prisma. Вместо того чтобы настраивать свой собственный хостинг, вы можете использовать Prisma Cloud, чтобы управлять всем, что касается DevOps.

- *AWS AppSync*. Еще одним новым игроком в экосистеме является сервис Amazon Web Services. Компания Amazon выпустила новый продукт, основанный на инструментах GraphQL и Apollo, чтобы упростить процесс настройки сервиса GraphQL. С помощью AppSync вы создаете схему, а затем подключаетесь к источнику данных. AppSync обновляет данные в режиме реального времени и даже обрабатывает их изменения в автономном режиме.

Сообщество Slack Channels

Отличный способ влиться в сообщество GraphQL — присоединиться к одному из его многочисленных каналов Slack. Вы сможете не только узнавать последние новости о GraphQL, но и задавать вопросы, на которые иногда отвечают создатели этих технологий.

Вы также можете поделиться своими знаниями с другими через следующие сообщества:

- GraphQL Slack, graphql-slack.herokuapp.com;
- Apollo Slack, www.apollographql.com/#slack.

По мере продолжения вашего путешествия по GraphQL вы можете стать более активным участником сообщества. Сегодня существуют такие общеизвестные проекты, как React Apollo, Prisma и сам GraphQL, которые публикуют статьи об актуальных проблемах с помощью тега `help wanted`. Ваша помощь в одном из этих вопросов может помочь многим! Существует также немало возможностей для внесения новых инструментов в экосистему.

Хотя изменения неизбежны, мы твердо стоим на земле, поскольку разработчики API GraphQL большие профессионалы. В основе всего, что мы делаем, — создание схемы и написание распознавателей для выполнения требований к данным схемы. Независимо от того, сколько инструментов потребовалось, для изменения чего-то в экосистеме мы можем опираться на стабильность самого языка запросов. API GraphQL хоть и новый, но будущее играет яркими красками. Итак, давайте же создадим нечто потрясающее.

Об авторах

Алекс Бэнкс (Alex Banks) и **Ева Порселло** (Eve Porcello) — инженеры-программисты и преподаватели. Живут в городе Тахо, штат Калифорния. Вместе с сотрудниками из своей компании Moon Highway они разработали и внедрили специальную учебную программу для корпоративных клиентов и онлайн-курсы на сайте LinkedIn Learning. Они также написали книгу *Learning React*, выпущенную издательством O'Reilly Media.

Об иллюстрации на обложке

Животное на обложке — орел Бонелли *(Aquila fasciata)*. Этот крупный хищник встречается в Юго-Восточной Азии, на Ближнем Востоке и в Средиземноморье. Предпочитает сухой климат и места, где он может гнездиться на скалах или высоких деревьях. Средний размах его крыльев составляет около 1,5 метра, у него темно-коричневая голова и крылья с белым подкрыльем, украшенным темными полосками и пятнами.

Бесшумный в полете, этот незаметный охотник питается в основном другими птицами, включая хищных, но поедает также мелких млекопитающих и рептилий. Несмотря на склонность к употреблению в пищу других хищных птиц, взрослые гнездящиеся пары известны своей привязанностью к птенцам независимо от происхождения и забирают яйца и птенцов из заброшенных гнезд как пернатых своего вида, так и хищных птиц других видов, для которых не характерна агрессия между родственниками.

Многие животные, изображенные на обложках O'Reilly, находятся под угрозой; все они важны для нашего мира. Чтобы узнать больше о том, как вы можете помочь, зайдите на сайт animals.oreilly.com.

Изображение для обложки создано Бремсом Тирлебеном (Brehms Tierleben).

Алекс Бэнкс, Ева Порселло

GraphQL: язык запросов для современных веб-приложений

Перевел с английского С. Черников

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Е. Рафалюк-Бузовская*
Художественный редактор	*С. Заматевская*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева,
д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 12.02.19. Формат 60×90/16. Бумага офсетная. Усл. п. л. 15,000.
Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87